

AD-A123 586

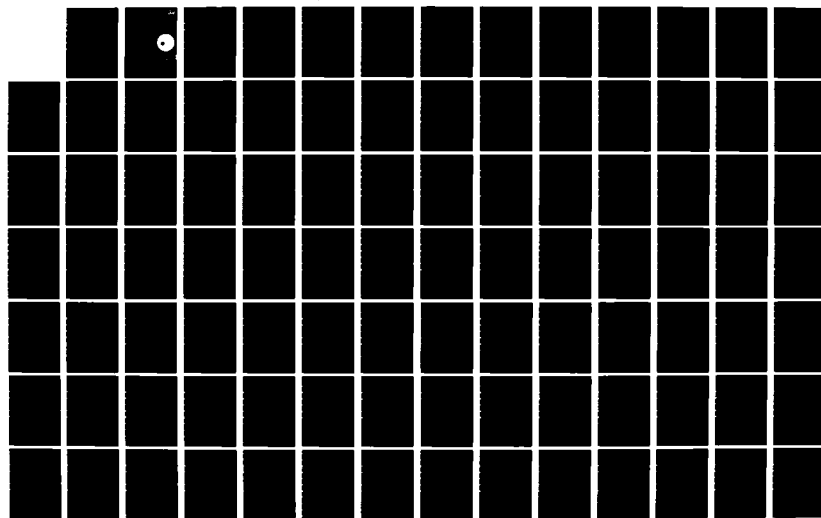
OPERATING SYSTEMS FOR RING-BASED MULTIPROCESSORS(U)
WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES
R A FINKEL ET AL. 1982 N00014-81-C-2151

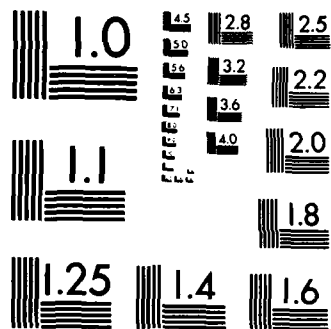
1/04

UNCLASSIFIED

F/G 9/2

NL





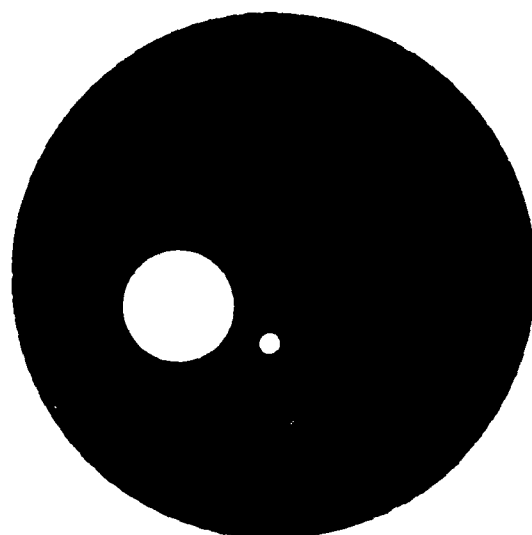
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



COMPUTER SCIENCES DEPARTMENT

University of Wisconsin-
Madison

AD-A123586



DTIC FILE COPY

FINAL REPORT

OPERATING SYSTEMS FOR RING-BASED MULTIPROCESSORS

Sponsored by

Defense Advanced Research Projects Agency (DoD)

ARPA Order No. 4095

Monitored by Naval Research Laboratory

Under Contract No. N00014-81-C-2151.

DTIC

ELECTE

MAY 20 1983

D

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

83 01 12 080



DEPARTMENT OF THE NAVY
NAVAL RESEARCH LABORATORY
WASHINGTON, D.C. 20375

IN REPLY REFER TO:
06 May 1983

Mr. J. E. Cundiff
Defense Technical Information Center (DTIC/DDAC)
Building 5
Cameron Station, Alexandria, VA 22314

Dear Sir:

Subj: Authorization to copy and distribute report

Ref: (a) Final Report, Operating Systems for Ring-Based Multiprocessors,
University of Wisconsin - Madison, Principal Investigators
Raphael A. Finkel and Marvin H. Solomon, ARPA Order No. 4095,
Contract No. N00014-81-C-2151, monitored by Naval Research
Laboratory

The appendices of reference (a) contain material that includes copyright notices. I have been assured by Prof. Finkel that at least one of the authors of each of the copyrighted articles was in fact supported completely or in part by Defense Department funds during the time the reported work was performed. Specifically, Drs. Leland and Fishburn were supported entirely by ARPA funds, and Profs. Finkel and Solomon were supported by ARPA funds for the work on which their names appear.

I have received the enclosed releases from Prof. Finkel. They grant permission to reproduce the material in appendices A, B, and E of the report (in the case of appendix E, an annotation is requested by the publisher). Appendices D, F, G, and H contain no copyright markings.

Consequently, I request that DTIC add the requested annotation to appendix E and delete appendix C prior to distributing the report. In place of Appendix C, you may use the following citation: "This material was published in Information Processing Letters, Vol. 12, No. 3 (13 June 1981), pp. 117-120, North Holland Publishing Co., Amsterdam."

Very truly yours,

Carl E. Landwehr
Scientific Officer

Encls - 3

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By <u>Per Ltr. on File</u>	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	



Copyright Release, per Ltr. on file

FINAL REPORT

OPERATING SYSTEMS FOR RING-BASED MULTIPROCESSORS

Sponsored by

Defense Advanced Research Projects Agency (DoD)

ARPA Order No. 4095

Monitored by Naval Research Laboratory

Under Contract No. N00014-81-C-2151.

December 1982

Contractor:

University of Wisconsin-Madison

750 University Ave.

Madison, WI 53706

Effective: 80 DEC 19

Expires: 82 JAN 19

Principal Investigators:

Raphael A. Finkel

Marvin H. Solomon

1210 W. Dayton St.

Madison, WI 53706

(608) 262-1204

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

CONTENTS

1. SUMMARY	1
2. THEORETICAL STUDIES	2
2.1. Process Migration	2
2.2. Processor Interconnection Strategies	3
2.3. Distributed Algorithms	5
3. OPERATING SYSTEMS IMPLEMENTATION	7
3.1. History	7
3.2. Enhancements to Arachne	8
3.3. Charlotte	9
4. REFERENCES	11
5. APPENDICES	
A: <i>Speedup of Distributed Algorithms</i> by J. P. Fishburn	
B: <i>Density and Reliability of Interconnection Topologies for Multicomputers</i> by W. E. Leland	
C: <i>High-Density Graphs for Processor Interconnection</i> by W. E. Leland et al	
D: <i>A Stable Distributed Scheduling Algorithm</i> by R. M. Bryant and R. A. Finkel	
E: <i>Parallelism in Alpha-Beta Search</i> by R. A. Finkel and J. P. Fishburn	
F: <i>The Arachne Kernel, Version 1.2</i>	
G: <i>Roscoe Utility Processes</i>	
H: <i>Arachne User Guide, Version 1.2</i>	

1. SUMMARY

This report summarizes the activities supported by DOD contract N00014-81-D-2151 (ARPA order number 4095) and the results of those activities. The period of this contract was originally scheduled to be from 80 Dec 19 through 81 Dec 19, but it was extended by a no-cost extension until 82 Jan 29. It was succeeded by DOD contract N00014-82-C-2087 (ARPA order number 4095) which supports a continuation of research activities in the same area. We view the former contract as supporting a pilot project, initiating a research program that is still under way. For this reason, the following pages will have more of the flavor of an interim report than a final summation.

A variety of research activities are supported by this project, all sharing the common theme of hardware and software architectures for distributed computing. A related DARPA-sponsored project at another institution is developing a very high-bandwidth communications medium (in the multi-gigabit/sec range) using LSI laser technology and fiber optics. Such a communications medium suggests the possibility of attacking computationally intensive tasks by connecting a large number (at least 10^3) of processing elements in an MIMD multicomputer. To exploit the fruits of such an organization, however, many difficult questions of high-quality system architecture, algorithmic design, and software engineering must be solved.

Our research may be grouped into two broad activities. First, we are carrying out a continuing project to construct and experiment with a prototype operating system for such a multicomputer. Second, we have pursued several theoretical investigations concerning system architecture and algorithmic design, testing our conclusions with a combination of mathematical analysis, simulation, and experimental implementation. We will describe first the theoretical studies and then the operating systems implementation activities.

2. THEORETICAL STUDIES

2.1. Process Migration

Two essential problems must be solved before process migration for load balancing becomes successful. The first problem is one of policy: When should a process be migrated? The second is one of implementation: How is a process migrated? Professor Finkel, in collaboration with Professor Ray Bryant, has investigated the first issue [1]. They developed a new scheduling algorithm for a multicomputer connected in a point-to-point fashion.

The algorithm is both distributed, in that every processor runs the same algorithm, and stable, in that collective load balancing decisions will not cause unnecessary overloading of a processor in the network. The migration algorithm assumes that any process may run equally well on any machine, that the cost of migration is proportional to the memory requirements of the process, that main memory is not a limiting constraint on any machine. Simulations were performed in Simpas [2] to test the algorithm.

The first part of the algorithm is to estimate the current load independently on each machine, so that adjacent machines with different loads can exchange a process. Load is estimated as the time it would take an average new process to complete on the machine, given that each process currently on the machine executes for a time equal to its execution time so far. Theoretical justification for this measure can be found in the report [1].

The second part of the algorithm is to find a neighbor that has a different load. Several versions of the pairing algorithm [3] were evaluated. This algorithm dynamically creates mutually agreeable pairings between adjacent machines to allow them to exchange a process.

The third part of the algorithm is to determine which processes, if any, should be sent. Each process on the heavier-loaded machine is considered a potential candidate for migration. Its completion time is estimated both for keeping it where it is and migrating it. That process with the best improvement in service ratio is migrated, and then selection repeats to find other processes that might profitably migrate as well.

Simulation results showed that even when the load of the network was fairly uniform, the migration algorithm significantly reduced the variance of the load and increased throughput. When the network was unbalanced, a condition maintained by directing all new arrivals to the same small subset of machines, migration was able to convert an instable situation into a stable one.

2.2. Processor Interconnection Strategies

For several years, the principal investigators have been investigating graph-theoretical questions concerning optimal topologies for interconnecting large numbers of processing elements [4,5]. Recently Will Leland, a graduate research assistant supported by the project, completed doctoral research in this area under the direction of Professor Solomon.

A large network of processing elements connected by point-to-point communications lines may be modeled by an undirected graph in which the vertices represent processing elements and the edges represent communications lines. Several figures of merit are possible for evaluating such a graph. If each processor is to be directly connected to every other, the graph must be K_n , the complete graph on n vertices. However, in such a graph, as the number n of vertices increases, the number of communications lines increases as n^2 , and the *degree* (number of neighbors of each vertex) increases as n . In many applications, one or the other of these costs is unacceptable. In particular, the design of the individual processing elements may limit the number of neighbors to a

constant d , independent of n . We may comply with such a restriction if we allow communication between processing elements to be indirect, through intermediate processing elements. In this case, we say the *distance* between two vertices is the number of edges (communications lines) in the shortest path connecting them, and the *diameter* of the graph is the largest distance between any two vertices.

Briefly stated, then, the problem is to find a graph with smallest possible diameter k for a given number n of vertices and a given bound d on the degree. For many years, an easily derived upper bound has been known for the number of vertices in a graph with diameter k and degree d : $N \leq 1 + \frac{d}{d-2}[(d-1)^k - 1]$. This bound can be derived from a very simple argument, the details of which make it seem likely that the bound is unduly optimistic. It has been proved that for $d \geq 3$ at most three graphs actually attain the bound (so the bound is too large by at least 1), and for degree 3, the bound is too large by at least 2 (for almost all diameters). Aside from these rather disappointing results, no better bound has ever been discovered.

This bound can be restated as a lower bound on diameter: $k \geq \log N / \log(d-1)$. Previous researchers have considered graphs in which the diameter varies as some power of the number of vertices. Others have proposed families of graphs, such as the *hypercube*, that require an increase of both d and k to increase N . The best previously announced infinite families of graphs for small fixed degrees are $2 \lg N$ for degree 3 (a complete binary tree achieves this bound) and $\lg N$ for degree 4 (achieved by the deBruijn graph), where "lg" denotes the base-2 logarithm. Solomon and Leland discovered a new family of degree-3 graphs with diameter $1.5 \lg N$. Leland was able to extend these ideas, improving the constant slightly and applying the ideas to larger diameters.

The preceding paragraphs consider the asymptotic behavior of the diameter as the number of vertices approaches infinity. Another approach is to attempt to find large graphs for certain specific values of k and d . Using various techniques, including computerized heuristic search, Leland was able to find many new large graphs with diameter and degree less than or equal to 10. His results were reported in [6].

Diameter is not the only figure of merit for an interconnection topology. Another important consideration is connectivity. Multiple paths between pairs of nodes are desirable, both to eliminate traffic bottlenecks, and to improve resistance to node failures. Leland considered the problem of connectivity from several points of view. He showed that the connectivity of his new family of graphs (in the graph-theoretical sense of minimal number of vertices that must be removed to partition the graph into two or more disconnected parts) was very good. He analyzed expected distribution of traffic among vertices under the assumption that the source and destination of a message were chosen randomly from a uniform distribution. Finally, he defined a new quantity, called the *average random failset size (ARFS)*, which is the expected size of a randomly-chosen set of vertices sufficient to partition the graph, and studied this quantity for various families of graphs.

2.3. Distributed Algorithms

Our work in distributed algorithms has addressed a general and important problem: How much speedup is possible with n machines running an algorithm that employs messages as the means of cooperation between machines? A recent PhD thesis by Jack Fishburn, under the supervision of Raphael Finkel [7], sheds light on the question by presenting and analyzing several parallel algorithms for multicomputers.

First, two distributed algorithms for α - β search are presented for trees of processors. Each processor is an independent computer with its own memory and is connected by communication lines to each of its nearest neighbors. The first algorithm, Tree Splitting, was implemented on Arachne and its behavior was measured. A formal analytical model was developed that predicts best-case, average, and worst-case speedup. In the worst case, speedup is $\sqrt{n}$ for n processors; in the best case, speedup is n . The second algorithm, Mandatory Work First, was also analyzed. Its behavior is bounded by fairly complex expressions, but for reasonable assumptions about the problem, speedup is about $n^{0.65}$. Related publications include [8, 7, 9, 10, 11, 12]

Next, numerical algorithms that exhibit a locally-defined iterative character were investigated. Such algorithms include solution of the Dirichlet problem. Natural multicomputer adaptations can be built for these algorithms. The thesis investigates two interconnection topologies, a grid and a tree. The results can be generalized to situations involving arbitrarily shaped domains of any dimensionality. Formal analysis derives the running time of the grid and tree algorithms with respect to per-message overhead, per-point communication time, and per-point computation time. The overall result is that the larger the problem, the closer the algorithms approach optimal speedup.

Finally, the thesis considers adaptations of large-network algorithms to small networks. A large-network algorithm requires n machines for a problem of size n . The thesis presents a general method for transforming large-network algorithms into quotient-network algorithms that solve problems of size n with fewer processors. The transformation allows algorithms to be designed assuming any number of processors. Implementing such algorithms on a quotient network results in no loss of efficiency, and often a great savings in hardware cost. This work has been published [13].

3. OPERATING SYSTEMS IMPLEMENTATION

3.1. History

The current project had its origins in several equipment grants from NSF (#MCS77-08968, #MCS78-06809, #MCS79-07516, and #MCS80-06499) for research in the application of distributed computer systems. With these funds we purchased five Digital Equipment LSI-11/03 computers and a larger PDP-11/40 software development host. We also acquired enough point-to-point communications interfaces to connect the 5 small machines to each other and to the development host. We developed the Roscoe operating system [14, 15, 16, 17] later called Arachne, for these machines. Roscoe was the first operational network computer operating system written explicitly for a local network. This project was supported in part by the Graduate Research Foundation of the University of Wisconsin, which provided summer support for the principal investigators and salaries for graduate research assistants.

More recently, NSF funds have been applied to upgrade both the processors and the interconnection hardware. We currently have 8 Digital Equipment PDP-11/23 computers connected using a communications interface called the *Megalink*, manufactured by Computrol Corporation of Danbury, Connecticut. These computers are not only about 2.5 times as fast as the previous ones, but also have memory management, allowing us to build significantly larger programs. The Megalink is a 1-megabit/second broadband contention bus with carrier-sense but not collision-detection capabilities.

More recently, the Computer Sciences Department has acquired a Digital Equipment VAX/780, which has begun to serve as our development host. Programs are cross-compiled on the VAX and are then loaded through the Megalink into the PDP-11 computers.

3.2. Enhancements to Arachne

We have brought Arachne to a stable state from which we can see how the hardware influenced our design choices and how they, in turn, led to the behavior of the entire operating system [19]. Locally-designed ROM bootstrap programs were developed and placed in each machine to simplify initial loading. We revised Arachne to use the new memory-management facility. This change was harder than it first appeared, since ability to access all memory directly was assumed in many places in the kernel. One immediate advantage was that more and larger programs could fit into each machine, since there is more physical memory and processes can share code. Another advantage is that processes and the kernel are protected from each other. However, Arachne does not do swapping, so no more processes can run on one machine than will fit in physical memory at once, and stacks and data areas cannot grow once a process has started.

With the move to a contention communications medium we had to deal with collisions. In one preliminary experiment, one processor "listens" and reports what it hears, while the remaining seven processors attempt to send it packets as fast as they can. Each sits in a tight loop waiting for the carrier to drop and then immediately sends a 4096-byte packet. We find that the "listener" hears about 100 packets at a time without errors, followed by a brief burst of packet collisions. Because the senders are all executing the same algorithm on the same hardware, there is a high degree of coupling, so that two senders who collide once will often collide several times in a row. In this sense, the experiment is a "worst-case" test. It seems to indicate that the absence of collision detection is not as severe as might be expected, since even under worst case, collisions are rare.

Nonetheless, since collisions are possible, we had to develop protocols. We divide our protocol architecture into two levels: intermachine and interprocess. The intermachine-level protocol provides reliable, in-order delivery of frames from one machine to another, avoiding recopying of data where possible, and allowing for recovery when frames fail to be delivered either because of temporary contention on the line, unavailability of the destination computer, or unwillingness of a recipient to receive the message. We use a simple alternating-bit stop-and-wait protocol because end-to-end delay is very short.

The interprocess-level protocol multiplexes several links on one intermachine connection. We debugged these protocols by simulating them in the Simpas event-driven simulation language [2]. This technique of debugging protocols was very natural; the protocols themselves are structured as actions triggered by asynchronous events.

3.3. Charlotte

While we were adapting Arachne to the new hardware, we have begun designing its successor, Charlotte, which builds on lessons learned from the implementation of Arachne. Charlotte differs from Arachne in several respects.

One difference is intended use of the operating system. Arachne was a pilot project to test our ideas for building multicomputer operating systems; Charlotte is intended as a production operating system to be used by a wide research community. A real user community tests more realistically the strengths and weaknesses of any software. To attract such a community Charlotte needs a variety of useful utility programs, including text editors, compilers, and command interpreters.

Another difference lies in the basic communications structure. Interprocess communication in Arachne is based on the notion of a *link*, which is a simplex communications path from one process to another. A client holding a link

to a server from which it desires service must create a link to itself and enclose it in a request, so that the server can respond. We found in Arachne that when two processes need to communicate they usually need a duplex communications path, even if the bulk of the information transfer is in one direction. Therefore, we decided to introduce the concept of a *duplex link* in Charlotte. Another motivation for duplex links is our intention to experiment with the use of process migration. A problem with simplex links as implemented in Arachne is that no information about the source end of the link is stored available at the destination end. This implementation interferes with process migration, since it is impossible for the operating system kernel of the destination process to find all potential senders to inform them about the move.

One of the design goals of Arachne was to take as much function as possible out of the kernel and put it into *utility processes*, which are treated like ordinary processes by the kernel, but which provide "systems" services such as file and other resource management. An important utility process in Arachne was the *Resource Manager*, which is responsible for allocating the processor resource. The kernel is responsible for providing the necessary implementation primitives for allocating memory and starting processes, while the Resource Manager is responsible for policy decisions such as which processor to start a new process on. Charlotte carries this trend further. The functions of the Resource Manager will be divided among a *Starter* process that initiates new processes, a *Switchboard* process that provides a directory service, registering and dispensing links to other utility processes, and a *Connector* process, which helps set up links with a multi-process applications program. These ideas will be described more fully in a forthcoming report.

A final difference between Arachne and Charlotte is implementation language. We were both pleased and disappointed with C as the implementation language for Arachne. On the positive side, it allowed us to be considerably

more productive that we could have been had we tried to implement Arachne in assembler language, while allowing sufficient flexibility to permit us to avoid use of assembler language almost entirely. On the negative side, the C compiler provides almost no assistance in catching runtime errors. Indeed, the design of the C language, with its heavy reliance on pointer variables, makes such things as subscript and pointer checking nearly impossible. We found that an undue amount of our time was spent tracking down errors due to subscript violations, dangling pointers, and incorrect numbers of arguments to procedures. These errors are particularly hard to find, since they may not manifest themselves until long after they occur, and since they may cause unrelated thoroughly debugged routines to malfunction. As a result of these experiences, we have decided to use Modula [20] as the implementation language for Charlotte. We have a locally-written compiler for Modula that is capable of producing code for variety of target machines including the PDP-11 and the VAX. Having our own locally written and locally maintained compiler allows us to modify the language to meet our particular needs and may allow experimentation with systems-implementation language design in the future.

During the first year of this project, the design of Charlotte was nearly completed and implementation is nearly ready to commence.

4. REFERENCES

- [1] R. M. Bryant and R. A. Finkel, "A Stable Distributed Scheduling Algorithm," *Proc. Second International Conference on Distributed Computing Systems*, pp. 314-323 (April 1981).
- [2] R. M. Bryant, "SIMPAS User Manual," Technical Report 391, University of Wisconsin-Madison Computer Sciences (June 1980).
- [3] R. A. Finkel, M. H. Solomon, and M. L. Horowitz, "Distributed algorithms for global structuring," *Proc. National Computer Conference 48*, AFIPS Press, pp. 455-460 (June 1979).
- [4] R. A. Finkel and M. H. Solomon, "Processor interconnection strategies," *IEEE Transactions on Computers C-29*, 5, pp. 360-371 (May 1980).
- [5] R. A. Finkel and M. H. Solomon, "The Lens Interprocessor Connection Strategy," *IEEE Transactions on Computers C-30*, 12, pp. 960-965 (December 1981).

- 1981).
- [6] W. E. Leland, R. A. Finkel, Li Qiao, M. H. Solomon, and L. Uhr, "High density graphs for processor interconnection," *Information Processing Letters* 12, 3, pp. 117-120 (13 June 1981).
 - [7] J. P. Fishburn, "Analysis of speedup in distributed algorithms," Technical Report 431, University of Wisconsin-Madison Computer Sciences (May 1981) Ph.D. thesis.
 - [8] J. P. Fishburn, "An optimization of alpha-beta search," *SIGART Newsletter*, 72, pp. 29-30 (July 1980).
 - [9] J. P. Fishburn and R. A. Finkel, "Parallel Alpha-Beta Search on Arachne," Technical Report 394, University of Wisconsin-Madison Computer Sciences (July 1980).
 - [10] R. A. Finkel and J. P. Fishburn, "Parallelism in Alpha-Beta Search," *Journal of Artificial Intelligence* 19, 1, (September 1982).
 - [11] J. P. Fishburn, R. A. Finkel, and S. A. Lawless, "Parallel alpha-beta search on Arachne," *Proc. 1980 International Conference on Parallel Processing*, pp. 235-243 (August 1980).
 - [12] R. A. Finkel and J. P. Fishburn, "Improved speedup bounds for parallel alpha-beta search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (February 1982) (accepted).
 - [13] J. A. Fishburn and R. A. Finkel, "Quotient networks," *IEEE Transactions on Computers* C-31, 4, (April 1981).
 - [14] M. H. Solomon and R. A. Finkel, "The Roscoe distributed operating system," *Proc. 7th Symposium on Operating Systems Principles*, pp. 108-114 (December 1979).
 - [15] M. H. Solomon and R. A. Finkel, "ROSCOE: A multi-microcomputer operating system," *Proc. 2nd Rocky Mountain Symposium on Microcomputers*, pp. 291-310 (August 1978).
 - [16] R. A. Finkel, M. H. Solomon, and R. Tischler, "Arachne User Guide, Version 1.2," Technical Summary Report 2066, University of Wisconsin Mathematics Research Center (April 1980).
 - [17] R. A. Finkel and M. H. Solomon, "The Arachne Kernel, Version 1.2," Technical Report 380, University of Wisconsin-Madison Computer Sciences (April 1980).
 - [18] R. A. Finkel, M. H. Solomon, and R. L. Tischler, "Roscoe Utility Processes," Technical Report 338, University of Wisconsin-Madison Computer Sciences (February 1979).
 - [19] R. A. Finkel and M. H. Solomon, "The Arachne Distributed Operating System," Technical Report 439, University of Wisconsin-Madison Computer Sciences (July 1981).
 - [20] N. Wirth, "Modula: A language for modular multiprogramming," *Software Practice and Experience* 7, 1, pp. 3-35 (1977).

APPENDIX A

Analysis of Speedup in Distributed Algorithms (Ph.D. Thesis)

John P. Fishburn

Technical Report 431
Computer Sciences Department
University of Wisconsin--Madison
May 1981

ANALYSIS OF SPEEDUP IN DISTRIBUTED ALGORITHMS

by

JOHN PHILIP FISHBURN

A thesis submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

MAY 1981

© Copyright by John Philip Fishburn 1981

All rights reserved

ANALYSIS OF SPEEDUP IN DISTRIBUTED ALGORITHMS

John Philip Fishburn

Under the supervision of

Assistant Professor Raphael A. Finkel

Abstract

We present and analyze several practical parallel algorithms for multicomputers.

Chapter four presents two distributed algorithms for implementing alpha-beta search on a tree of processors. Each processor is an independent computer with its own memory and is connected by communication lines to each of its nearest neighbors. Measurements of the first algorithm's performance on the Arachne distributed operating system are presented. For each algorithm, a theoretical model is developed that predicts speedup with arbitrarily many processors.

Chapter five shows how locally-defined iterative methods give rise to natural multicomputer algorithms. We consider two interconnection topologies, the grid and the tree. Each processor (or terminal processor in the case of a tree multicomputer) engages in serial computation on its region and communicates border values to its neighbors when those values become available. As a focus for our investi-

gation we consider the numerical solution of elliptic partial differential equations. We concentrate on the Dirichlet problem for Laplace's equation on a square region, but our results can be generalized to situations involving arbitrarily shaped domains (of any number of dimensions) and elliptic equations with variable coefficients. Our analysis derives the running time of the grid and the tree algorithms with respect to per-message overhead, per-point communication time, and per-point computation time. The overall result is that the larger the problem, the closer the algorithms approach optimal speedup. We also show how to apply the tree algorithms to non-uniform regions.

A large-network algorithm solves a problem of size N on a network of N processors. Chapter six presents a general method for transforming large-network algorithms into quotient-network algorithms, which solve problems of size N on networks with fewer processors. This transformation allows algorithms to be designed assuming any number of processing elements. The implementation of such algorithms on a quotient network results in no loss of efficiency, and often a great savings in hardware cost.

ACKNOWLEDGMENTS

Thanks go first to Raphael Finkel, who as my adviser has spent an enormous amount of energy editing rough drafts and prodding me in fruitful directions.

Jim Goodman, Will Leland, Diane Smith, Marvin Solomon, John Strikwerda, and Len Uhr have provided many helpful discussions.

Karl Anderson got me started on parallel alpha-beta search by directing me to Baudet's work.

The Graduate School provided fellowship support for the past year.

I am grateful to these people and thank them all.

Finally, I want to give special thanks to Sharon Lawless for writing part of the Arachne checkers program, Mary Leland for providing Lemma 4.5, and Raphael Finkel for translating this thesis into English.

TABLE OF CONTENTS

Chapter 1 - Introduction.....	1
Chapter 2 - Judging Parallel Algorithms.....	4
Chapter 3 - Previous Research.....	7
3.1. SORTING AND MERGING.....	7
3.1.1. Comparison-exchange networks.....	7
3.1.2. Parallel tape sorting.....	8
3.1.3. Multi-processor methods.....	9
3.1.4. SIMD methods.....	10
3.1.5. Vector sorting.....	10
3.2. NUMERICAL METHODS.....	10
3.2.1. Fast-Fourier transform.....	11
3.2.2. Adaptive Quadrature.....	11
3.2.3. Matrix methods.....	12
3.3. GLOBAL STRUCTURING.....	13
3.4. GRAPH THEORY.....	14
Chapter 4 - Parallel Alpha-Beta Search.....	16
4.1. INTRODUCTION.....	15
4.2. THE ALPHA-BETA ALGORITHM.....	17
4.3. RELATED WORK.....	23
4.3.1. Parallel-Aspiration Search.....	24
4.3.2. Mandatory-Work-First Search.....	25
4.4. THE TREE-SPLITTING ALGORITHM.....	25
4.4.1. The Leaf Algorithm.....	26
4.4.2. The Interior Algorithm.....	27
4.4.3. Alpha Raising.....	30
4.5. MEASUREMENTS OF THE ALGORITHM.....	31
4.6. OPTIMIZATIONS	34
4.7. ANALYSIS OF SPEEDUP.....	36
4.7.1. Worst-first ordering.....	38
4.7.2. Best-first ordering.....	39
4.7.3. Discussion.....	46
4.7.4. Random Order.....	48
4.7.5. Discussion of Theorem 4.6.....	61
4.8. MANDATORY-WORK-FIRST SEARCH.....	63

4.8.1.	Best-First Order.....	69
4.8.2.	Worst-First Order.....	72
4.8.3.	Other Orderings.....	75
4.9.	COMPARISON OF PALPHABETA AND MWF.....	77
4.10.	TIPS FOR PROCESSOR-TREE ARCHITECTS.....	78
4.10.1.	Serial versus Parallel.....	79
4.10.2.	Maximal Processor Trees.....	82
Chapter 5 - Piecewise-Serial Iterative Methods.		84
5.1.	INTRODUCTION.....	84
5.2.	THE DIRICHLET PROBLEM.....	85
5.2.1.	Jacobi Method.....	86
5.2.2.	Gauss-Seidel Method.....	87
5.2.3.	Successive Over-Relaxation.....	88
5.3.	PREVIOUS WORK.....	88
5.4.	PIECEWISE-SERIAL ITERATIVE METHODS.....	91
5.4.1.	Uniform Regions With Grid Topology.....	91
5.4.2.	Uniform Regions With Tree Topology.....	94
5.4.3.	Efficiency.....	110
5.4.4.	Measurement of Communication Time.....	111
5.4.5.	Scheduling Tree Machines.....	114
5.4.6.	Static Non-Uniform Regions.....	116
5.4.7.	Dynamic Region Encroachment.....	123
Chapter 6 - Quotient Networks.....		127
6.1.	INTRODUCTION.....	127
6.2.	EXISTING NETWORKS.....	128
6.2.1.	Grid-Connected Network.....	128
6.2.2.	Perfect Shuffle.....	129
6.2.3.	PM2I.....	130
6.2.4.	Cube.....	130
6.3.	EXISTING ALGORITHMS.....	130
6.3.1.	Fast-Fourier Transform on the Shuffle..	131
6.3.2.	Sorting on the Shuffle.....	134
6.3.3.	Polynomial Evaluation on the Shuffle...	134
6.3.4.	Finite-difference Methods.....	135
6.4.	NETWORK EMULATION.....	135
6.4.1.	Perfect Shuffle.....	137
6.4.2.	Grid-connected Network.....	139
6.4.3.	Cube.....	142
6.4.4.	PM2I.....	144
6.5.	SOME RESULTING ALGORITHMS.....	146
6.5.1.	Fast-Fourier Transform on the Shuffle..	146
6.5.2.	Sorting on the Shuffle.....	148
6.5.3.	Polynomial Evaluation on the Shuffle...	149
6.5.4.	Finite-difference Methods.....	149

6.5.5.	Alpha-beta Search.....	149
6.6.	THE ECONOMICS OF EMULATION.....	150
Chapter 7 - Conclusions and Future Directions..		152
7.1.	ALPHA-BETA SEARCH.....	152
7.2.	PIECEWISE-SERIAL ITERATIVE METHODS.....	154
7.3.	QUOTIENT NETWORKS.....	156
7.4.	PARALLEL PROGRAMMING PROVERBS.....	157
7.4.1.	Large Computation per Message.....	157
7.4.2.	Do Interesting Work First.....	158
7.4.3.	Do Mandatory Work First.....	159
7.4.4.	Do Something.....	159
Appendix A - Some Optimizations of α - β Search..		161
A.1.	FALPHABETA.....	161
A.2.	LALPHABETA.....	165
A.3.	CALPHABETA.....	167
A.4.	MEASUREMENTS.....	168
REFERENCES.....		171

Chapter 1 - Introduction

Three helping one another will do as
much as six men singly.

- Spanish Proverb

SHUTTLE DELAYED; COMPUTERS WOULDN'T TALK

- headline, The Capital Times
10 April 1981

Most computers consist of a single central processing

unit (CPU), a store of words, and a communications line between them. A program's task is to change the store's content in some significant way. It accomplishes this change by passing information along the line, one word at a time, back and forth between the CPU and store. Because of its serial nature, Backus [1] calls this line the "von Neumann bottleneck".

The von Neumann bottleneck not only limits the speed of ordinary computers, but also forces us to think of algorithms in serial terms. In recent years, several computer architectures have been proposed that avoid the von Neumann bottleneck by allowing many computations to proceed simultaneously. Some of these architectures have been built and are working [2,3,4,5,6,7].

We use the taxonomy of Flynn [8] to divide parallel processors into two broad classes: MIMD and SIMD. In the

MIMD (Multiple Instruction stream, Multiple Data stream) model, each processor is a separate computer with its own program counter. Each processor computes independently of all others. MIMD computers can be broken down into subclasses: A multicomputer consists of several ordinary computers connected only by communications lines. Arachne [7] is a multicomputer. A multiprocessor consists of several CPUs with shared access to a common memory. C.mmp [3] is a multiprocessor.

In an SIMD network, a central controller broadcasts one instruction at a time to all the processors in the network, which then execute the instruction simultaneously on their own data. Illiac IV [2] is an SIMD network.

If parallel architectures are to ever become widely useful, we must learn how to use them efficiently. In this thesis we investigate the use of parallel architectures in performing certain computations. We assume throughout that each processing unit has a private memory and is connected by communications lines to some of the other processors. Communication is restricted to data passed on these lines; no shared memory exists in our model. Although we usually assume the multicomputer model, Chapters 5 and 6 deal with interconnection networks that may be SIMD or MIMD.

In Chapter 2 we discuss figures of merit for parallel algorithms. Chapter 3 briefly surveys previous work in the

field of parallel algorithms. In Chapter 4 we present two parallel alpha-beta search algorithms. The alpha-beta pruning technique is used by programs that play games like chess to speed up the search of the tree of possible continuations. Alpha-beta search presents a challenge to the designer of parallel algorithms because of its inherently serial nature: Results from searching one part of the lookahead tree reduce the computation for searching another part. If both searches proceed independently, these savings are reduced.

Chapter 5 presents several parallel implementations of the Jacobi method. The Jacobi method is an important technique for numerically solving certain partial differential equations such as the Dirichlet problem.

A large-network algorithm solves a problem of size N on an interconnection network of N processors. In Chapter 6 we present a general method for transforming large-network algorithms into quotient-network algorithms, which solve problems of size N on networks with fewer processors. This transformation allows algorithms to be designed for certain interconnection topologies assuming any number of processing elements.

In Chapter 7 we summarize the contributions of this thesis and discuss areas for further work.

Chapter 2 - Judging Parallel Algorithms

When the judge is unjust he is no longer a judge but a transgressor.

- Phillips Brooks
Visions and Tasks

By what standards can we judge parallel algorithms?

The most commonly used gauge of a parallel algorithm's performance is speedup, which we define as:

$$\text{speedup} = \frac{\text{time required by the best serial algorithm}}{\text{time required by the parallel algorithm}}$$

The running time for the parallel algorithm includes time required for data movement. Sometimes speedup is of overwhelming importance. For example, if a 24-hour weather-prediction program that runs serially in 48 hours could be made to run four times faster with a tenfold increase in hardware, such a conversion might very well be considered appropriate. Whenever someone must wait for a program to complete, we may be willing to pay for more than an N -fold increase in hardware to obtain an N -fold speedup. Examples of such computations are database transactions and work performed for interactive users.

Another criterion used to judge parallel algorithms is efficiency:

$$\text{efficiency} = \frac{\text{speedup}}{\text{number of processors used}}$$

The use of this criterion assumes that cost is proportional to the number of processors. This assumption is sometimes

overoptimistic, as for example in architectures that use a crosspoint switch of complexity N^2 to connect N processors to N memories.

Sometimes the efficiency of a parallel algorithm is greater than one, which leads us to conclude that the serial algorithm to which it is compared is not the best available. For example, Baudet [9] found, for k equal to two or three, more than k -fold speedup in performing alpha-beta search with k processors. However, the serial algorithm under comparison unwisely started the search with the window $(-\infty, +\infty)$, instead of the usual narrow window. (This algorithm is described in detail in Chapter 4.)

The designer of parallel algorithms hopes to achieve an efficiency of one. Some algorithms achieve this (e.g. Pease's use of the perfect shuffle [10] to compute FFTs), others come close (The efficiency of Batcher's sorting algorithm [11] is $1/\log N$), others fall short (Csanky's algorithm [12] computes the inverse of a matrix with efficiency $1/N \log^2 N$).

A third criterion is practicality, by which we mean the likelihood that the required hardware will exist at some time in the near future. Many of the algorithms reviewed in Chapter 3 are quite impractical. First, many of these algorithms assume that all processors have equal and

non-interfering access to a common memory. Workable hardware that fits this description is both expensive and inefficient, especially when the number of processors is large. (The best example is C.mmap [3], which uses an expensive crossbar switch to connect processors to memory, and has serious problems with memory contention.) Common memory places a practical upper bound on the number of processors that can be used. In effect, the von Neumann bottleneck on a shared memory machine must serve many processors instead of one. Second, many parallel algorithms require N (or worse, N^2) processors to solve a problem of size N .

The average Ph.D. thesis is nothing but a transference of bones from one graveyard to another.

- J. Frank Dobie

The body of literature on parallel algorithms is rapidly growing. Most of it assumes a C.mmp-like (MIMD, shared-memory) architecture. In this section we review previous results in parallel algorithms for sorting, numerical methods, global structuring, and graph theory.

3.1. SORTING AND MERGING

Previous work in parallel sorting methods can be divided into five broad categories: comparison-exchange networks, parallel tape sorting, multiprocessor methods, SIMD methods, and vector sorting.

3.1.1. Comparison-exchange networks

A comparison-exchange network accepts N numbers on N input lines and sorts them onto N output lines by means of a network of comparison-exchange modules. After receiving two numbers on its two input lines, a comparison-exchange module always places the larger number on a particular output line and the smaller on the other output line. A sort-

ing network, as we shall call it, can also be thought of as a multiprocessor architecture with the following nonadaptivity constraint: Whenever two numbers K_i and K_j are compared, the subsequent comparisons in the case $K_i < K_j$ are identical to those in the case $K_i > K_j$, except that i and j are interchanged. Batcher, in one of the earliest results on sorting networks [11], shows how to sort N words in $(1/2)\log N(\log N + 1)$ steps with approximately $(1/2)\log^2 N$ ranks of $N/2$ modules each. Stone [13], building on Batcher's work, accomplishes the same task with only one rank of modules. Muller [14], by allowing the network to contain AND and OR gates and single-pole, double-throw switches, as well as the comparator modules, shows how to sort N words in time $O(\log N)$. Unfortunately, this scheme uses $O(N^2)$ elements. Knuth [15] gives an excellent, but dated, review of sorting networks.

3.1.2. Parallel tape sorting

Parallel tape sorting is a parallel version of external tape sorting. Parallel tape sorting exploits the fact that, after information has been placed on a tape by one computer, the tape can be used immediately as input to another computer. This technique is unique among parallel sorting methods in not assuming special hardware. Given N records, $\log N$ processors, and $4(\log N)$ tapes, Even [16]

gives a method that sorts the records onto a tape in time $3N^2$, where it is assumed that a record can be read from tape and written onto another tape in one time unit.

3.1.3. Multi-processor methods

Multi-processor methods use the architectural model of many processors with equal access to a common memory. As with most algorithms, most of the work done in parallel sorting schemes assumes this architecture. Valiant [17] gives an algorithm for sorting N words with N processors in $2\log N \cdot \log \log N + O(\log N)$ comparison steps. Gavril [18] gives an algorithm that merges two linearly ordered sets of size N , with $P \leq N$ processors in $2[\log(N+1)] + (4N/P)$ steps. Hirschberg [19] gives an algorithm that sorts N words in time $O(K \log N)$ using $N^1 + 1/k$ processors, for k an arbitrary integer. Unfortunately, the amount of memory necessary is MN , where the numbers to be sorted are in the range $[0, M-1]$. Furthermore, memory fetch conflicts do exist. That is, the algorithm assumes an architecture that, in one time unit, can satisfy multiple read requests to a single memory cell. Preparata [20] improves on Hirschberg's algorithms by giving a family with identical performance, but without memory fetch conflicts.

3.1.4. SIMD methods

Baudet [21] gives an algorithm for sorting N words on K processors in time $(N \log N)/K + O(N)$. Hence, when $K \ll \log N$, the speedup is optimal in the number of processors used. Baudet's method can thus be considered practical, since performance can be boosted linearly with a small number of processors. Thompson [22], on the other hand, gives two algorithms for sorting N^2 words on an N -by- N mesh-connected processor array (like the Illiac IV) in $O(N)$ routing and comparison steps.

3.1.5. Vector sorting

Stone [23] studies several different sorting methods for one particular architecture, the CDC STAR computer. He shows that although the $N(\log^2 N)$ computational complexity of Batcher's bitonic sorting algorithm is worse than Quicksort's, the bitonic sort's good use of STAR's vector instructions allows it to out-perform Quicksort on vectors of reasonable size.

3.2. NUMERICAL METHODS

In this section, we will review parallel numerical algorithms that have been developed for the fast-Fourier transform, zero-finding, adaptive quadrature, and matrix

computations. A review of parallel algorithms for finite-difference calculations is given in Chapter 5.

3.2.1. Fast-Fourier transform

One of the most important discoveries in algorithms in recent years has been the fast-Fourier transform (FFT) [24]. Pease [10] demonstrates that the perfect shuffle interconnection pattern can yield optimal speedup in the computation of the DFT. Specifically, he shows that $\log N$ passes through $N/2$ multiply-add modules, alternating with $\log N$ passes through an N -line shuffle-exchange, is sufficient to compute the DFT of N points in time $O(\log N)$. (We discuss both the FFT and the shuffle-exchange network in detail in Chapter 6.)

Flanders [6] shows how to accomplish the necessary routing for the computation of a DFT on a rectangular grid of processors.

3.2.2. Adaptive Quadrature

Lemne [25] describes a parallel architecture for calculating finite-sum estimates of one-dimensional integrals. The architecture consists of a tree of computers connected by communications lines. In addition, all leaf processors have access to a large common memory that specifies a queue of tasks for each processor. These queues are managed by a

set of queue-balancing processors. The speedup associated with this configuration is shown to be at least $O(N/\log N)$ with N processors.

3.2.3. Matrix methods

Parallel algorithms have been developed for solving tridiagonal systems, band triangular systems, and for matrix inversion and matrix multiplication.

Traub [26] and Stone [27] both consider the solution of equations whose matrix has nonzero elements only on the three central diagonals. Traub proposes an iterative method, called Parallel Gauss, and shows that m processors can solve a linear system of size m in time $O(1)$. The Parallel Gauss method can be run on SIMD machines (like the Illiac IV), or on C.mmp-like machines. Stone [28] presents an iterative method, called the odd-even reduction algorithm, that under diagonal-dominance conditions converges more quickly than Traub's parallel Gauss method.

Chen [29] considers the solution of lower-triangular systems of equations. Chen gives direct methods for solving these systems, and shows that when the bandwidth of the matrix is $m+1$ (diagonals further than m away from the main diagonal are all zero), these methods yield a speedup of approximately p/m with p processors.

Gentleman [30], by considering only data movement, gives lower bounds to the parallel complexity of certain matrix operations. In particular, he shows that for machines with two-dimensional rectangular grid connectivity (like the Illiac IV), multiplication and inversion of N -by- N matrices inherently require $O(N)$ steps.

Csanky [12] gives an algorithm that computes the inverse of an N -by- N matrix in time $O(\log^2 N)$ using $O(N^4)$ processors. Preparata [31], by modifying Csanky's algorithm, shows that the same time bound can be achieved with $2N^{4/3}/(\log^2 N)$ processors if multiplication of two N -by- N matrices can be done in parallel in time $O(\log N)$ using $N^4/\log N$ processors.

3.3. GLOBAL STRUCTURING

Pinkel [32] empirically investigates distributed algorithms for imposing global structure on graphs. Pairing algorithms match together in pairs as many neighboring nodes as possible. Spanning tree algorithms select a subset of the edges that provides a unique path between any two nodes. Finally, developing hierarchies generalizes the pairing algorithms in two ways: First, nodes are organized into groups of arbitrary size, rather than in pairs. Second, these groups are then treated as nodes, to be

grouped into meta-groups, and so on. These algorithms assume the graph to be identical to the physical network topology. Moreover, information about the structure of the graph is itself distributed: Each processor knows only of its own immediate neighbors. These algorithms are therefore suitable not for processing arbitrary graphs on arbitrary networks, but for organizing a given physical network as a basis for solving other problems.

Chang [33] assumes similar ground rules, and addresses the tasks of finding a minimal spanning tree of a weighted graph, distributing a list, and finding the extrema of a set of nodes, given that their names obey a total ordering.

3.4. GRAPH THEORY

Most work in parallel graph-theoretic algorithms assumes an architectural model of many computers with equal access to a large common memory. Given an adjacency matrix as input, Hirschberg [34] gives a parallel algorithm that uses N^2 processors to compute the connected components of an undirected graph with N nodes in time $O(\log^2 N)$. Savage [35] presents a family of algorithms that use $O(\log^2 N)$ running time, and polynomial-in- N processors, to solve the following problems for a graph with N nodes: For a connected, undirected graph G , find a spanning tree, a cycle,

a cycle basis, the bridges and bridge connected components, and the biconnected components of G . For a connected, undirected, weighted graph, find a minimum spanning tree. For a connected, directed graph, find a cycle, a shortest cycle, the dominators, and the dominator tree.

Chapter 4 - Parallel Alpha-Beta Search

The axe is already laid at the root of the trees; so every tree that fails to yield good fruit will be cut down and thrown into the fire.

- John the Baptist

By the Nine Gods he swore it,
And named a trysting day,
And bade his messengers ride forth
East and west and south and north,
To summon his array.

- "Lays of Ancient Rome"
Lord Macaulay

4.1. INTRODUCTION

The α - β search algorithm is central to most programs that play games like chess. It is now well-known [36] that an important component of the playing skill of such programs is the speed at which the search is conducted. For a given amount of computing time, a faster search allows the program to "see" farther into the future. In this Chapter we present and analyze two parallel adaptations of the α - β algorithm. The first adaptation, which we call the tree-splitting algorithm, speeds up the search of a large tree of potential continuations by dynamically assigning subtree searches for parallel execution.

In section 2, we review the q - p algorithm. Section 3 discusses parallel implementations of the q - p algorithm suggested by other workers. Section 4 formally describes the tree-splitting algorithm. Section 5 presents performance measurements for this algorithm taken on a network of microprocessors. Section 6 discusses some possible optimizations and variations of the algorithm. Section 7 derives the obtainable speedup with k processors as k tends towards ∞ .

The second adaptation, mandatory work first, is a formalization of a method proposed by Akl, Barnard, and Doran [37]. Section 8 analyzes this algorithm. Section 9 compares it with the tree-splitting algorithm. We close with a number of suggestions for architectural design of processor trees.

4.2. THE ALPHA-BETA ALGORITHM

Consider a board position from a game like chess or checkers. All possible sequences of moves from this position may be represented by a tree of positions called the lookahead tree (Figure 4.1). The nodes of the tree represent positions; the children of a node are moves from that node. The root node of the tree represents the current position. Since lookahead trees for most games are

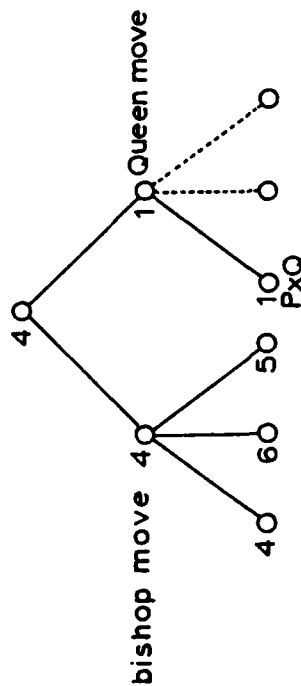


Fig. 4.1. Lookahead tree.

often too large to be searched even by computer, they are usually truncated at a certain level. Since we will later be referring to a tree of processors, we reserve the following notation for nodes of lookahead trees: A node is often called a position. A node's child is its successor, and its parent is its predecessor. If each interior node has n successors, we say that the tree has degree n . The level of a node or subtree is its distance from the root.

The α - β algorithm is an optimization of the minimax algorithm, which we will review first. The two players are called max and min; at the root node, it is max's turn to move. The minimax algorithm proceeds as follows: First, each leaf of the lookahead tree is assigned a static value that reflects that position's desirability. (High values are desirable to max. In a game like chess, the main component of the value is usually the material balance between the two sides.)

The interior nodes of the lookahead tree may be given minimax values recursively: If it is max's turn to move at node A, the value of A is the maximum of A's successors' values. If the game were to proceed to node A, it would then be max's turn to move. Max, being rational, would choose the successor with the maximum value, say M. Therefore, the subtree rooted at A must have M as its value, because M is the value of the leaf node we would reach if the

game reached A. Similarly, if it is min's turn to move at a node, then the value of that node is the minimum of these values.

We will use a version of the minimax procedure called negamax: When it is max's turn to move at a terminal node, the node is assigned the same static value used in minimax. When it is min's turn to move, the static value assigned is the negative of what it would be in the minimax case. The value of an interior node at any level is defined to be the maximum of the negatives of the values of its successors.

The negamax algorithm can be cast into an ad hoc Pascal-like language. The following program is adapted from Knuth [38] :

```

function negamax(p: position): integer;
var m: integer;
    i, d: 1..MAXCHILD;
    succ: array[1..MAXCHILD] of position;
begin
    determine the successor positions
      succ[1], ..., succ[d];
    if d = 0 then { terminal node }
      negamax := staticvalue(p)
    else
      begin { find maximum of child values }
        m := -oo;
        for i := 1 to d do
          m := max(m, - negamax(succ[i]));
        return(m);
      end
end

```

end.

The α - β algorithm evaluates the lookahead tree without pursuing irrelevant branches. Suppose we are investigating the successors in a game of chess, and the first move we look at is a bishop move. After analyzing it, we decide that it will gain us a pawn. Next we consider a queen move. In considering our opponent's replies to the queen move, we discover one that can irrefutably capture the queen; she has moved to a dangerous spot. We need not investigate our opponent's remaining replies; in light of the worth of the bishop move, the queen move is already discredited.

The α - β search algorithm [38] formalizes this notion:

```

function alphabeta(p : position;  $\alpha, \beta$  : integer) : integer;
var i, d : 1..MAXCHILD;
    succ : array[1..MAXCHILD] of position;
begin
    determine the successor positions
        succ[1], ..., succ[d];
    if d = 0 then
        alphabeta := staticvalue(p)
    else
        begin
            for i := 1 to d do
                begin
                     $\alpha$  := max( $\alpha$ , - alphabeta(succ[i], - $\beta$ , - $\alpha$ ));
                    if  $\alpha \geq \beta$  then return( $\alpha$ ) { cutoff }
                end;
            end;
        end;
end;

```

```

return( $\alpha$ );
end

```

The function alphabeta obeys the accuracy property:

For a given position p, and for values of α and β such that $\alpha < \beta$,

```

if negamax(p)  $\leq \alpha$ , then alphabeta(p,  $\alpha, \beta$ )  $\leq \alpha$ 
if negamax(p)  $\geq \beta$ , then alphabeta(p,  $\alpha, \beta$ )  $\geq \beta$ 
if  $\alpha < \text{negamax}(p) < \beta$ , then alphabeta(p,  $\alpha, \beta$ ) = negamax(p)

```

The first and second cases above are called falling low and falling high respectively. In the third case, success, alphabeta accurately reports the negamax value of the tree. Success is assured if $\alpha = -\infty$ and $\beta = \infty$. The pair (α, β) is called the window for the search.

To return to our example: When alphabeta is called with p representing the queen move, it is min's move. β is the cutoff value generated by the bishop move. The better the bishop move was for max, the lower is β . (Within the routine alphabeta, high values for α and β are good for the player whose move it is. A high value for α indicates that a good alternative for that player exists somewhere in the tree. A low value for β indicates that a good alternative exists for the other player somewhere else in the tree.) When the successor that captures the queen is evaluated, α becomes larger than β , and a cutoff occurs.

q - β pruning serves to reduce the branching factor, which is the ratio between the number of nodes searched in a tree of height N and one of height $N-1$, as N tends to ∞ . Both theory [38] and practice [39] agree that with good move ordering (investigating best moves first), q - β pruning reduces the branching factor from the degree of the lookahead tree nearly to the square root of that degree. For a given amount of computing time, this reduction nearly doubles the depth of the accessible lookahead tree.

When the algorithm is performed on a serial computer, the value of one successor can be used to save work in evaluating its siblings later on. Nevertheless, greater speed can be obtained by conducting q - β search in a parallel fashion.

We will restrict our attention to parallel computers built as a tree of serial computers. A node in this tree is a processor, the parent of a node is its master, and the child of a node is its slave.

4.3. RELATED WORK

In this section we review previous research in parallel alpha-beta algorithms.

4.3.1. Parallel-Aspiration Search

In order to introduce parallelism, Baudet [9] rejects decomposition of the lookahead tree in favor of a parallel aspiration search, in which all slave processors search the entire lookahead tree, but with different initial q - β windows. These windows are disjoint, and in the simplest variant their union covers the range from $-\infty$ to $+\infty$. Since each window is considerably smaller than $(-\infty, +\infty)$, each processor can conduct its search more quickly. When the processor whose window contains the true minimax value of the tree finishes, it reports this value, and move selection is complete. Baudet analyzes several variants of this algorithm under the assumption of randomly distributed terminal values, and concludes that the obtainable speedup is limited by a constant independent of the number of processors available. This maximum is established to be approximately 5 or 6. Surprisingly, for k equal to 2 or 3, Baudet's method yields more than k -way speedup with k processors. Baudet infers that the serial q - β search algorithm is not optimal, and estimates that a 15 to 25 percent speedup may be gained by starting the search with a narrow window.

Since a narrow window does not speed up a successful search when moves are ordered best-first, Baudet's method yields no speedup under best-first move ordering.

4.3.2. Mandatory-Work-First Search

Akl, Barnard, and Doran [37] distinguish between those parts of a subtree that must be searched and those parts whose need to be searched is contingent upon search results in other parts of the tree. By searching mandatory nodes first, their algorithm attempts to achieve as many of the cutoffs seen in the serial case as possible. This technique leads to an algorithm that we discuss in detail in Section 4.8.

4.4. THE TREE-SPLITTING ALGORITHM

A natural way to implement the q - p algorithm on parallel processors divides the lookahead tree into its subtrees at the top level and queues them for parallel assignment to a pool of slave processors. Each processor computes the value of its assigned subtree by using either serial q - p search (if it is a leaf processor) or parallel q - p search (if it has slaves of its own). When it finishes, it reports the value computed to its master. As a master receives responses from its slaves, it narrows its window and tells working slaves about the improved window. When all subtrees have been evaluated, the master is able to compute the value of its position.

4.4.1. The Leaf Algorithm

The leaf algorithm runs at leaf nodes of the processor tree. We will describe its interactions with its master by means of remote procedure calls. The algorithm can also be expressed in a message-passing or shared-memory form. The master calls the function `leafqp` (line 19) remotely. A master can interrupt a search in progress to tell its slave of a newly-narrowed window by invoking the asynchronous "update" procedure in the slave (line 3). The variables q and p (line 1) are global arrays, not formal parameters, in order to facilitate updating their values in each recursive call of `alphabeta` when the new window arrives.

Here is the leaf algorithm:

```

1  $q, p$  : array[1..MAXDEPTH] of integer;
3 asynchronous procedure update(newd, newp : integer);
4 { update is called asynchronously by my master
5   to inform me of the new window (newd, newp) }
6 var tmp : integer;
7   k : 1..MAXDEPTH;
8 begin
9   for k := 1 to MAXDEPTH do
10    begin { update  $q, p$  arrays }
11       $q[k]$  :=  $\max(q[k], \text{newd})$ ;
12       $p[k]$  :=  $\min(p[k], \text{newp})$ ;
13      tmp := newd;
14      newd := -newp;
15      newp := -tmp;
16    end

```

```

17 end;

19 function leafq(p : position; d, β : integer) : integer;
20 begin
21   q[1] := d;
22   β[1] := β;
23   return(alpha_beta(p, 1));
24 end;

26 function alpha_beta(p:position; depth:integer): integer;
27 var succ:array[1..MAXCHILD] of position; {successors}
28   succno : 1..MAXCHILD; { which successor }
29   succlim : 1..MAXCHILD; { how many successors }
30 begin
31   determine the successors succ[1], ..., succ[succlim];
32   if succlim = 0 then return(staticvalue(p));
33   for succno := 1 to succlim do
34     begin { evaluate each successor }
35       q[depth+1] := - β[depth];
36       β[depth+1] := - q[depth];
37       q[depth] := max(q[depth],
38         -alpha_beta(succ[succno], depth+1));
39       if q[depth] ≥ β[depth] then
40         return(q[depth]); { cutoff occurs }
41     end { for succno }
42   return(q[depth]);
43 end; { function alpha_beta }

```

4.4.2. The Interior Algorithm

The interior algorithm $\text{interiorq}\beta$ runs on interior nodes of the processor tree. When $\text{interiorq}\beta$ is activated, it generates all successors of the position to be evaluated

(line 25). Each of its slaves is requested to evaluate one of these positions; the remaining positions are queued for later service. Newly-narrowed windows are relayed to slaves by use of "update" calls (line 3).

The master may take various actions when its slave returns. First, if the returned value causes the current q value to increase, then the master sends $-q$ as an updated β value to all of its active slaves (line 39). Second, if q has been increased so that it becomes greater than or equal to β , then an q - β cutoff occurs. The nonpositive-width window is sent to all active slaves, quickly terminating them (line 39). Meanwhile, the master empties its queue of waiting successor positions. (In the algorithm shown below, this effect is achieved by invoking slaves with negative-width windows.) Third, if the queue of unevaluated successor positions is non-empty, the reporting slave is assigned the next position from the queue.

When all successors have been evaluated, the master returns the final value to its master. In a game situation, the algorithm at the root node might serve as the user interface, and would remember which move has the maximum value.

Here is the interior algorithm:

```

1 var gld, glp : integer; { global d, p }
2   q : integer; { depth of processor tree }
3 asynchronous procedure update(newd, newp : integer);
4 { update is called asynchronously by my master
5 to inform me of the new window (newd, newp) }
6 begin
7   atomically do
8     begin
9       gld := max(gld, newd);
10      glp := min(glp, newp);
11    end; { atomically do }
12    parfor all slaveid do
13      slaveid.update(-glp, -gld);
14    end; { update }
15
16 function interiorq(p: position; d, p: integer) : integer;
17 var succ: array[1..MAXCHILD] of position; { successors }
18 succno : 1..MAXCHILD; { which successor }
19 succlim : 1..MAXCHILD; { how many successors }
20 tmp : array[1..MAXCHILD] of integer;
21 function g : integer;
22 begin
23   gld := d;
24   glp := p;
25   determine the successors succ[1], ..., succ[succlim];
26   if succlim = 0 then return(staticvalue(p));
27   if depth(succ[1]) < q then
28     g := interiorq(p);
29   else g := leafq;
30   parfor succno := 1 to succlim do
31     begin
32       when slaveid := idleslave() do
33         tmp[succno] :=

```

```

34   -slaveid.g(succ[succno], -glp, -gld);
35   if tmp[succno] > gld then
36     begin
37       atomically do gld := max(tmp[succno], gld);
38       for all slaveid do
39         slaveid.update(-glp, -gld);
40     end; { if tmp[succno] > gld }
41     return(gld);
42   end; { parfor succno }
43 end; { interior }

```

4.4.3. Alpha Raising

As an optimization of the interior algorithm, the master running on the root node may send a special α - β window to a slave working on the last unevaluated successor. This window is $(-\alpha-1, -\beta)$ instead of the usual $(-\beta, -\alpha)$. If that successor is not the best, then the slave's search will fail high as usual, but the minimal window speeds its search. If that successor is best, then the smaller window causes the search to fail low, again terminating faster. In either case, the root master determines which successor is the best move, even though its value may not be calculated. By speeding the search of the last successor, the idle time of the other slaves is reduced. (This narrow window given to the root's last subtree search can also be used in serial α - β search as discussed in the appendix.)

We can generalize this technique in the following way, called alpha raising: Suppose that each successor of the root is being evaluated by a different slave, and that slave₁'s current q value, q_1 , is lower than any other, and that slave₂ has the second lowest q value, say q_2 . Update q_1 to q_2-1 , speeding up slave₁. If this update causes slave₁'s otherwise successful search to fail low, then the reported value is still lower than all others, and that move is still discovered to be best.

4.5. MEASUREMENTS OF THE ALGORITHM

Measurements of the performance of the tree-splitting algorithm have been taken on a network of LSI-11 microcomputers running under the Arachne [7] operating system.

The game of checkers was used to generate lookahead trees. Static evaluation was based on the difference in a combination of material, central board position for kings and advancement for men. Moves were ordered best-first according to their static values. General q -raising was not employed except for the special case for the last successor.

A single LSI-11 machine searches lookahead trees at a rate of about 100 unpruned nodes per second. Inter-machine messages can be sent at a rate of about 70 per second.

Only 5 processors were available in Arachne at the time of these experiments, so it was not possible to directly test processor trees of height greater than one. An estimate of the speedup of a tree of height two was made by exploiting the following fact: Since a master spends most of its time waiting for its slaves to finish their assigned tasks, the speed of a master is proportional to the speed of its slaves. One way to speed up a leaf processor is to replace it with a processor tree of height one. Therefore we can roughly equate the speedup of a height-two processor tree in searching a height- x lookahead tree with the product $Y_0 Y_1$, where Y_0 is the speedup of a height-one processor tree in searching a height- x lookahead tree, and Y_1 is the speedup of a height-one processor tree in searching a lookahead tree of height $x-1$.

Ten board positions, $B_1, \dots, B_{10}$, were chosen for use in these experiments. These positions actually arose during a human-machine game; they span the entire game. All lookahead trees from these positions were expanded to a depth of 8.

Two sets of experiments were performed. The two differed only in that the first set used one master and two slaves, while the second set used one master and three slaves. Within each experiment, Y_0 was measured directly for each B_i by evaluating the tree both serially and with

the parallel algorithm running on a depth-one processor tree. Table 1 summarizes measurements of γ_0 .

The ten board positions gave rise to 84 successors, so 84 EVALUATE commands were given to slaves while γ_0 was being measured. These 84 commands were saved, and times for both parallel and serial evaluation were measured for each command. The aggregate speedup for a group of commands is the total time required to execute them serially divided by the total time required to execute them in parallel. For each B_i , the aggregate speedup γ_i for its subtree evaluations was computed. Table 2 summarizes measurements of γ_i .

Table 1: γ_0 for each B_i , $i=1, \dots, 10$

	2 slaves	3 slaves
minimum	1.37	1.37
average	1.81	2.34
maximum	2.36	3.15
standard deviation	0.31	0.56

Table 2: γ_i for each B_i , $i=1, \dots, 10$

	2 slaves	3 slaves
minimum	1.03	1.38
average	1.46	1.96
maximum	1.77	2.60
standard deviation	0.22	0.38

Surprisingly, more than k-way speedup was occasionally achieved with k slaves: Three out of the ten B_i were sped up by more than 2 with 2 slaves, and two of those three were sped up by more than 3 with 3 slaves. Of the 84 subtrees of the B_i s, 4 were sped up by more than 2 with 2 slaves, and 9 were sped up by more than 3 with 3 slaves; 2 of those achieved 6-way speedup. In each such case, subtree evaluations finished in a different order than they were assigned. While one large subtree was being evaluated by one slave, another smaller subtree was assigned and finished. The large subtree's evaluation then received an UPDATE message that sped it up or even terminated it. In fact, time-consuming searches are more likely than short ones to receive these messages. In particular, the search that receives the final ($-d-1, -d$) window is likely to be larger than average.

4.6. OPTIMIZATIONS

Since the tree-splitting algorithm can be optimized in several ways, it should be considered the simplest variant of a family of tree-decomposing algorithms for α - β search. As a first optimization, since most of a master's time is spent waiting for messages, that time could be spent profitably doing subtree searches. However, only the deepest

masters could hope to compete with their slaves in conducting searches. All other masters are by themselves slower than their slaves because their slaves have slaves below them to help. However, more than half of all masters control leaf processors, and greater speedup should be achieved by running a leaf algorithm along with these masters on the same processors. We might expect an additional 1.5-way speedup from this technique.

A second optimization groups several higher-level masters onto a single processor. For example, the 3 highest processors in a binary processor tree could be replaced by 3 processes running on a single processor.

Third, a master might evaluate a position by assigning that position's successor's successors to slaves, rather than that position's successors. Although this technique involves more message-passing, some advantage might result, because all of a master's slaves would work on finishing the position's first subtree before going on to the second. The evaluation of the second subtree would then receive the full benefit of the beta value generated by the first subtree. Furthermore, when slaves become idle as one subtree is finished, they can immediately be set to work on the next subtree.

Since most game-playing programs must make their move within a certain time limit, any speedup in tree search

ability will generally be used to search a deeper lookahead tree. If we have an unlimited supply of processors to form into a binary tree, we can obtain an unlimited speedup only if the search is not limited in time. Otherwise we cannot, because we would eventually violate our premise that the lookahead tree is at least as deep as the processor tree. A new layer on the processor tree does not buy another full ply in the lookahead tree. For example, several speedups of 1.5 would be needed to search a 6-times larger chess lookahead tree, or about one additional ply. The depth of the processor tree would grow faster than the depth of the tree it searches and eventually would catch up. The only way to avoid this limit is to increase the fan-out of the processor tree. If the fan-out is high enough that no successor need ever be queued for evaluation by a slave, then the size of the maximum lookahead tree that can be evaluated within the time limit is limited only by the time required for EVALUATE commands to propagate from the root to the leaves. Long before this limitation is reached, we would run out of silicon for making the processors.

4.7. ANALYSIS OF SPEEDUP

We now turn to a formal analysis of the speedup that can be gained in searching large lookahead trees as the

number of available processors grows without bound. For this purpose we introduce Palphabeta, a simplified version of the tree-splitting algorithm. This algorithm is in general less efficient than the version already discussed, but is more amenable to analysis. Much of the analysis in this section is a "parallelization" of results of Knuth [38]. Indeed, when $q = 0$ and $f = 1$, Theorem 1 and Corollary 1 reduce to Knuth's results.

As before, the processors will be arranged in a uniform tree. Let $f \geq 1$ be the fan-out of the processor tree (uniform for all interior nodes), and let $q \geq 1$ be its depth (uniform for all terminal nodes). Let $q + s$ be the depth of the lookahead tree, where $s \geq 1$. We assume that the lookahead tree has a uniform degree and that this degree, df , is a multiple of f , where d is ≥ 2 . Here is Palphabeta:

```

1 function Palphabeta(p:position; d,β,integer): integer;
2 var i : integer;
3 function g : integer;
4 begin
5   determine the successors p1, ..., pdf;
6   if depth(p1) < q then
7     g := Palphabeta
8   else g := alphabeta;
9   for i := 1 to d do
10    begin
11      d := max(d, max
                (i-1) f < j ≤ i · f
                -g(pj, -β, -d));

```

```

12   if d ≥ β then return(d);
13 end;
14 return(d);
15 end;

```

The f calls to function g in line 11 are intended to occur in parallel, activating functions existing on each of the f slaves. Serial q - β search is activated on leaf slaves; Palphabeta is activated on all others. Unlike the tree-splitting algorithm, Palphabeta waits until all slaves finish before assigning additional tasks. However, the two algorithms behave identically when searching either a best-first or worst-first ordered "theoretical" tree of uniform degree and depth. When we restrict ourselves to one of these lookahead trees, we can therefore make conclusions about the behavior of the tree-splitting algorithm by studying Palphabeta.

4.7.1. Worst-first ordering

q - β search produces no cutoffs if, whenever the call alphabeta(p, d, β) is made, the following relation holds among the successors $p_1, \dots, p_d$:

$$d < -\text{negamax}(p_1) < \dots < -\text{negamax}(p_d) < \beta.$$

We call this ordering worst first. If no cutoffs occur, it is easy to calculate the time necessary for Palphabeta to finish. Assume that a processor can generate f successors, send messages to all of its f slaves and receive replies in

time p . (This figure counts message overhead time but does not include computation time at the slaves.) Assume also that the serial q - β algorithm takes time n to search a lookahead tree with n terminal positions. Let a_n be the time necessary for a processor at distance n from the leaves to evaluate its assigned position. A leaf processor executes the serial algorithm to depth s . Thus we have $a_0 = (df)^s$. An interior processor gives d batches of assignments to its slaves, and each batch takes time p plus the time for the slave processor to complete its calculation. Thus we have $a_{n+1} = d(p + a_n)$. The solution to this recurrence relation is

$$a_q = p \left(\frac{d^{q+1} - d}{d - 1} \right) + d^q a_0^s,$$

which is the total time for Palphabeta to complete. Since the time for the serial algorithm to examine the same tree is $(df)^{q+s}$, the speedup for large s is f^q . There are $(f^{q+1}-1)/(f-1)$ processors, roughly f^q , so when no pruning occurs the parallel algorithm yields speedup that is roughly equal to the number of processors used.

4.7.2. Best-first ordering

We will now investigate what happens when the lookahead tree is ordered best-first.

Definition. We will use the Devey decimal system to name nodes in both processor trees and lookahead trees. The root is named by the null string. The j successors of a node whose name is $a_1 \dots a_k$ are named by $a_1 \dots a_k 1$ through $a_1 \dots a_k j$.

Definition. We say that the successors of a position $a_1 \dots a_n$ are in best-first order if

$$\text{negamax}(a_1 \dots a_n) = -\text{negamax}(a_1 \dots a_n 1).$$

Definition. We say a position $a_1 \dots a_n$ in the lookahead tree is (q, f) -critical if a_i is (q, f) -restricted for all even values of i or for all odd values of i . An entry a_i is (q, f) -restricted if

$$1 \leq i \leq q \text{ and } 1 \leq a_i \leq f$$

$$\text{or } q < i \text{ and } a_i = 1.$$

Theorem 4.1. Consider a lookahead tree for which the value of the root position is not $\pm \infty$ and for which the successors of every position are in best-first order. The parallel q - β procedure Palphabeta examines exactly the (q, f) -critical positions of this lookahead tree.

Proof. We will call a (q, f) -critical position $a_1 \dots a_n$ a type 1 position if all the a_i are (q, f) -restricted; it is of type 2 if a_j is its first entry not (q, f) -restricted and $n-j$ is even; otherwise (that is, when $n-j$ is odd), it is of type 3. Type 3 nodes have a_n (q, f) -restricted. The fol-

lowing statements can be established by induction on the depth of the position p . (Text in brackets refers to positions of depth $< q$.)

(1) A type 1 position is examined by calling $[P]alphanbeta(p, +\infty, -\infty)$. If it is not terminal, its successor position $[s]$ $p_1, p_2, \dots, p_f$ is $[are]$ of type 1, and $F(p) = -F(p_1) \neq \pm \infty$. This $[These]$ successor position $[s]$ is $[are]$ examined by calling $[P]alphanbeta(p_1, -\infty, +\infty)$. The other successor positions $p_2, \dots, p_{df}$ $[p_{f+1}, \dots, p_{df}]$ are of type 2, and are all examined by calling $[P]alphanbeta(p_i, -\infty, F(p_1))$.

(2) A type 2 position p is examined by calling $[P]alphanbeta(p, -\infty, \beta)$, where $-\infty < \beta \leq F(p)$. If it is not terminal, its successor $[s]$ $p_1, p_2, \dots, p_f$ is $[are]$ of type 3, and $F(p) = -F(p_1)$. This $[These]$ successor position $[s]$ is $[are]$ examined by calling $[P]alphanbeta(p_1, -\beta, +\infty)$. Since $F(p) = -F(p_1) \geq \beta$, cutoff occurs, and $[P]alphanbeta$ does not examine the other successors $p_2, \dots, p_{df}$ $[p_{f+1}, \dots, p_{df}]$.

(3) A type 3 position p is examined by calling $[P]alphanbeta(p, \alpha, +\infty)$ where $F(p) \leq \alpha < +\infty$. If it is not terminal, each of its successors p_1 is of type 2, and they are all examined by calling $[P]alphanbeta(p_1, -\infty, -\alpha)$. All of these searches fail high.

It follows by induction on the depth of p that the (q, f) -critical positions, and no others, are examined.

Q.E.D.

Figure 4.2 shows the best-first lookahead tree of degree four and depth four that is examined by Palphabeta running on a processor tree of fanout two and depth two. Corollary 4.1. If every position on levels $0, 1, \dots, q+s-1$ of a lookahead tree of depth $q+s$ satisfying the conditions of Theorem 4.1 has exactly df successors, for d some fixed constant, and for f the constant appearing in Palphabeta, then the parallel procedure Palphabeta (along with alphabeta, which it calls), running on a processor tree of fan-out f and height q , examines exactly

$$f \lfloor q/2 \rfloor (df) \lceil (q+s)/2 \rceil, f \lceil q/2 \rceil (df) \lfloor (q+s)/2 \rfloor - f^q$$

terminal positions.

Proof. There are $f \lfloor q/2 \rfloor (df) \lceil (q+s)/2 \rceil$ sequences $a_1, \dots, a_{q+s}$ with $1 \leq a_i \leq df$ for all i , such that a_1 is (q, f) -restricted for all even values of i . There are $f \lceil q/2 \rceil (df) \lfloor (q+s)/2 \rfloor$ such sequences with a_1 (q, f) -restricted for all odd values of i . We subtract f^q for the sequences $\{1, \dots, f\}^q$, that we counted twice.

Q.E.D.

Lemma 4.1. Given positive constants a, b, c, d , and p , the relations

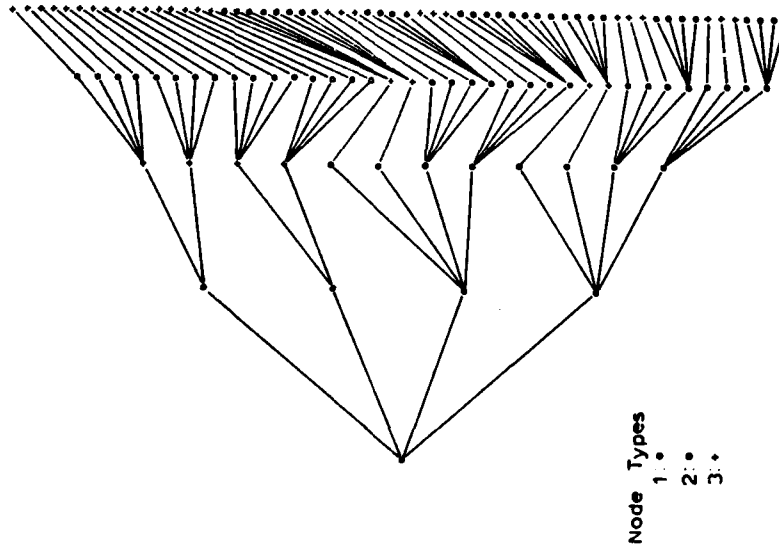


Fig. 4.2. Lookahead tree examined by Palphabeta.

$$a_0 = a; \quad a_{n+1} = pd + a_n + (d-1)b_n;$$

$$b_0 = b; \quad b_{n+1} = p + c_n;$$

$$c_0 = c; \quad c_{n+1} = d(p + b_n).$$

are satisfied by the sequences

$$a_n = \begin{cases} (n \text{ even:}) & a + h(n) [d(3p+bc) + p-b-c] - np, \\ (n \text{ odd:}) & a + h(n-1) [d(3p+bc) + p-b-c] - np \\ & + d(n-1)/2 (d(p+b) + p-b); \end{cases}$$

$$b_n = \begin{cases} (n \text{ even:}) & p + 2pg(n) + (p+b)d^{n/2}, \\ (n \text{ odd:}) & p + 2pg(n+1) + cd(n-1)/2, \end{cases}$$

$$c_n = \begin{cases} (n \text{ even:}) & 2pg(n+2) + cd^{n/2}, \\ (n \text{ odd:}) & 2pg(n+1) + (p+b)d^{(n+1)/2}, \end{cases}$$

where the function g is defined by

$$g(n) = (d^{n/2} - d)/(d - 1),$$

and the function h is defined by

$$h(n) = (d^{n/2} - 1)/(d - 1).$$

Proof. straightforward algebra.

Theorem 4.2. Under the conditions of Corollary 4.1, and assuming also that (1) serial d - p search is performed in time equal to the number of leaves visited, and (2) in p units of time, a processor can generate f successors of a position, send a message to each of its f slaves, and receive the f replies, the total time for Palphabeta to complete is

$$(q \text{ even:}) \quad (df) \lfloor s/2 \rfloor + (df) \lceil s/2 \rceil - 1$$

$$+ h(q) [d(3p + (df) \lfloor s/2 \rfloor + (df) \lceil s/2 \rceil) + p - (df) \lfloor s/2 \rfloor - (df) \lceil s/2 \rceil] - pq,$$

$$\begin{aligned} & (q \text{ odd:}) (df) \lfloor s/2 \rfloor + (df) \lceil s/2 \rceil - 1 \\ & + h(q-1) [d(3p + (df) \lfloor s/2 \rfloor + (df) \lceil s/2 \rceil) + p - (df) \lfloor s/2 \rfloor - (df) \lceil s/2 \rceil] - pq \\ & + d(q-1)/2 [d(p + (df) \lfloor s/2 \rfloor + p - (df) \lfloor s/2 \rfloor) + p - (df) \lfloor s/2 \rfloor]. \end{aligned}$$

Proof. Let a_n , b_n , and c_n represent the time required for a processor at distance n from the leaves of the processor tree to search type 1, 2, and 3 positions, respectively.

Then these sequences satisfy the relations

$$\begin{aligned} a_0 &= (df) \lfloor s/2 \rfloor + (df) \lceil s/2 \rceil - 1, \quad a_{n+1} = pd + a_n + (d-1)b_n; \\ b_0 &= (df) \lfloor s/2 \rfloor, \quad b_{n+1} = p + c_n; \\ c_0 &= (df) \lceil s/2 \rceil, \quad c_{n+1} = d(p + b_n). \end{aligned}$$

By substituting the constant expressions for a_0 , b_0 , and c_0 to find a_q by the formulas given by Lemma 4.1, we obtain the desired formula.

Q.E.D.

Under conditions of best-first search, the parallel α - β algorithm gives $O(k^{1/2})$ speedup with k processors for searching large lookahead trees. The next theorem formalizes this result:

Theorem 4.3. Suppose that Palphabeta runs on a processor tree of depth $q \geq 1$ and fan-out $f > 1$. Suppose that the lookahead tree to be searched is arranged in best-first

order and is of degree df and depth $q+s$, where $d \geq 1$. Denote by R the time for alphabeta to search this tree, and by P the time for Palphabeta to search the tree. Then

$$\lim_{s \rightarrow \infty} R/P = f^{q/2}$$

Proof. The time for the serial algorithm is

$$(df) \lfloor (s+q)/2 \rfloor + (df) \lceil (s+q)/2 \rceil - 1,$$

from Corollary 4.1. If we divide this quantity by the expression given by Theorem 4.2 for P , and take the limit as s goes to ∞ , we obtain the desired result.

Q.E.D.

4.7.3. Discussion

The improvement that alphabeta search shows over negamax search is due to the cutoffs it achieves. Parallel execution tends to lose some of that advantage, since subtrees that the serial algorithm would avoid are searched before information is available to cut them off. This situation is most extreme if the lookahead tree is ordered best-first; in this case the serial algorithm enjoys the most cutoffs. However, our analysis shows that even in this case, $O(k^{1/2})$ speedup can still be expected. At the other extreme, if the lookahead tree is ordered worst-first, then no cutoffs are found in either the serial or the parallel algorithm. In this case, the parallel algo-

rithm performs no wasted work, and speedup is $O(k)$.

We can now compare the measurements presented in Section 5 with these theoretical bounds. If we take $\gamma_0 \sqrt{1}$ to be the speedup achieved by a processor tree of depth two, then the measured speedup for height-two processor trees of fan-out two and three is 2.64 and 4.59 respectively. Table 3 summarizes theoretical best-first, theoretical worst-first, and measured speedups for processor trees of height one and two, and of fan-out two and three.

Table 3: Speedup

g	f	worst-first	best-first	measured
1	2	2	1.41	1.81
1	3	3	1.73	2.34
2	2	4	2.00	2.64
2	3	9	3.00	4.59

In checkers, certain simplifying assumptions used for the analysis are not true. The lookahead tree is neither regular nor ordered best- (nor worst-) first. Therefore, slave processors do not finish in unison. Nonetheless, our implementation results with checkers display speedups that lie between the two analytically derived extremes. Although tests with more processors should be run, these limited results show that the formal analyses are not unreasonable.

4.7.4. Random Order

Under best-first and worst-first ordering of uniform lookahead trees, sibling slaves finish simultaneously because each slave's pruned lookahead tree has the same size and shape. This fact makes it possible to calculate, for a given processor tree and lookahead tree, the exact finishing time for the algorithm Palphabeta. In this section, we analyze the behavior of the slightly weaker algorithm Pbound (no deep cutoffs) under the assumption that terminal values are independent, identically distributed random variables. Restated, this assumption says that no two terminal values are equal, and that any one of the $n!$ orderings of the terminal values is as likely as any other. Although the expected finishing times for sibling slaves are identical, the finishing times themselves may be unequal. Pbound must therefore wait for the last busy slave to finish before assigning the next batch of tasks. For this reason, we will not attempt to calculate the expected finishing time for the parallel algorithm under conditions of random ordering of terminal nodes. We will, however, present a "parallel" version of Knuth's [38] analysis of the serial algorithm under conditions of random order. The analyses of the parallel and serial cases both yield estimates of the expected number of terminal positions examined. Only in the serial case, however, does

this estimate yield a direct estimate of the finishing time of the algorithm.

Here is parallel α - β search without deep cutoffs:

```

function Pbound(p : position ; limit : integer) : integer ;
var m,i,t,d : integer ;
begin
  determine the successors  $p_1, \dots, p_d$ ;
  m := - $\infty$ ;
  if depth( $p_1$ ) < q then fn := pbound else fn := bound;
  for i := 1 to d do
    begin t :=  $\max_{1 \leq j \leq i} -fn(p_j, -m)$ ;
      if t > m then m := t;
      if m  $\geq$  limit then return(m);
    end;
  return(m);
end;

```

Pbound is called with limit = ∞ on the root node of the lookahead tree. On leaf processors, Pbound activates the serial algorithm without deep cutoffs:

```

function bound(p : position ; limit : integer) : integer ;
var m,i,t,d : integer ;
begin
  determine the successors  $p_1, \dots, p_d$ ;
  if d = 0 then return(staticvalue(p)) else
    begin m := - $\infty$ 
      for i := 1 to d do
        begin t := -bound( $p_i, -m$ );
          if t > m then m := t;
          if m  $\geq$  limit then return(m);
        end;
    end;

```

```

return(m);
end;
end;

```

Let $T(d,h)$ be the number of terminal positions examined by bound(p,q) in a tree rooted at p of depth h and degree d with randomly distributed terminal values.

Knuth [38] establishes that $T(d,h)$ satisfies

$$c_1(d)r_1^h \leq T(d,h) \leq c_2(d)r_2^h,$$

where c_1 and c_2 depend on d but not h, and r_1 and r_2 satisfy $c_3 d / \ln d \leq r_1$ and $r_2 \leq c_4 d / \ln d$, for certain constants c_3 and c_4 . As part of the proof of this result, the inequality

$$(4.1) \quad \left(\sum_{1 \leq i \leq d} \left(\sum_{1 \leq j \leq d} i^{-t((j-1)/2d)} \right)^{s/t} \right)^{1/s} \leq c_4 d / \ln d$$

is established for a certain choice of s, t satisfying $1/s + 1/t = 1$.

We begin by presenting a lemma due to Knuth and then adapting it to our own use.

Lemma 4.2. Suppose that $Y_{1,1}, \dots, Y_{i-1,d}$ and $Z_1, \dots, Z_{j-1}$ are independent sequences of (i-1)d and (j-1) independent identically distributed random variables.

Then
$$\frac{1}{\left(1 - 1 + \frac{(j-1)}{i-1}\right)/d}$$

is the probability that

$$(4.2) \quad \max_{1 \leq k < i} (\min(Y_{k,1}, \dots, Y_{k,d})) < \min_{1 \leq k < j} Z_k$$

Proof. If $i = 1$, the left hand side is $-\infty$. If $j = 1$, the right hand side is $+\infty$. In both cases, the probability that the relation holds is 1.

Assume then that $i, j > 1$. Consider the minimum element Y_{k_1, t_1} , over all $1 \leq k_1 < i$ and $1 \leq t_1 \leq d$. The probability that it is less than $\min_{1 \leq k < j} Z_k$ is

$$\frac{(i-1)d}{((i-1)d + j - 1)}.$$

Removing the elements $Y_{k_1, 1}, \dots, Y_{k_1, d}$ from consideration, we consider the minimum of the remaining Y s on the left of (4.2), say Y_{k_2, t_2} . The probability that Y_{k_2, t_2} is less than the right-hand side of (4.2) is

$$\frac{(i-2)d}{((i-2)d + j - 1)},$$

and so on. Hence (4.2) happens exactly when $Y_{k_1, t_1} < \text{RHS}$ and $Y_{k_2, t_2} < \text{RHS}$ and ... and $Y_{k_{i-1}, t_{i-1}} < \text{RHS}$, so (4.2) has probability

$$\begin{aligned} & \frac{(i-1)d(i-2)d \dots 1d}{((i-1)d + j - 1)((i-2)d + j - 1) \dots (d + j - 1)} \\ &= \frac{(i-1)!((j-1)/d)!}{((i-1) + (j-1)/d)!} \\ &= \frac{1}{\binom{i-1 + (j-1)/d}{i-1}} \end{aligned}$$

Q.E.D.

Lemma 4.3. (Corollary to Lemma 4.2): If $Y_{1,1}, \dots, Y_{(i-1)f, df}$ and $Z_1, \dots, Z_{(j-1)f}$ are independent sequences of $((i-1)f)df$ and $(j-1)f$ independent identically distributed random variables, then the probability P_{ij} that

$$\max_{1 \leq k \leq (i-1)f} \min_{1 \leq m \leq df} Y_{k,m} < \min_{1 \leq k \leq (j-1)f} Z_k$$

is

$$P_{ij} = \frac{1}{\binom{(i-1)f + (j-1)f/d}{(i-1)f}}$$

Proof. This Lemma is simply Lemma 4.2 with a change of variables.

Substitute:

$$\begin{aligned} df & \text{ for } d, \\ (i-1)f + 1 & \text{ for } i, \\ (j-1)f + 1 & \text{ for } j. \end{aligned}$$

Q.E.D.

Since the simple formula k^x is always within 12% of

$$\binom{k-1+x}{k-1},$$

for $0 \leq x \leq 1$ and k a positive integer [38], we will approximate P_{ij} by

$$(4.3) \quad P_{ij} = ((i-1)f + 1)^{-(j-1)/d}.$$

Theorem 4.4. Let $T(d, f, h)$ be the expected number of terminal positions examined by the parallel d - β procedure without deep cutoffs on a processor tree of degree f and

height h in a random uniform lookahead tree of degree df and height h . Then

$$T(d, f, h) < f^h c(d, f) r(d, f)^h,$$

where $r(d, f)$ is the largest eigenvalue of the matrix

$$M_{d, f} = \begin{pmatrix} \sqrt{p_{11}} & \sqrt{p_{12}} & \dots & \sqrt{p_{1d}} \\ \sqrt{p_{21}} & \sqrt{p_{22}} & \dots & \sqrt{p_{2d}} \\ \vdots & \vdots & \ddots & \vdots \\ \sqrt{p_{d1}} & \sqrt{p_{d2}} & \dots & \sqrt{p_{dd}} \end{pmatrix}$$

and $c(d, f)$ is a constant. The quantities p_{ij} in $M_{d, f}$ were defined in Lemma 4.3.

Proof. As before, assign Dewey decimal names to the positions of the lookahead tree. Define the functions

$$G(n) = \lfloor (n-1)/f \rfloor + 1$$

and

$$H(n) = \lfloor (n-1)/f \rfloor + 1.$$

The n th successor position is a member of the $G(n)$ th batch of successor positions to be assigned to slaves. The first member of that batch is the $H(n)$ th successor position.

When P bound examines position $a_1 \dots a_{m-1}$, "limit" is

$$\min_{1 \leq k < H(a_m)} \text{negamax}(a_1 \dots a_{m-2}k),$$

so its successor $a_1 \dots a_m$ is examined if and only if

$a_1 \dots a_{m-1}$ is examined and

$$\begin{aligned} & \min_{1 \leq k < H(a_m)} \text{negamax}(a_1 \dots a_{m-1}k) \\ & < \min_{1 \leq k < H(a_{m-1})} \text{negamax}(a_1 \dots a_{m-2}k) \end{aligned}$$

Abbreviate this inequality by P_m . Then $a_1 \dots a_h$ is examined if and only if $P_1, P_2, \dots$, and P_h hold. P_m holds with probability p_{ij} , where $i = G(a_{m-1})$ and $j = G(a_m)$. Furthermore, P_m is a function of the terminal values

$$\text{staticvalue}(a_1 \dots a_{m-2} j k b_{m+1} \dots b_n)$$

for

$$1 \leq j < H(a_{m-1}) \text{ and all } 0 \leq k, b \leq df$$

or

$$j = H(a_{m-1}) \text{ and } 1 \leq k < H(a_m) \text{ and all } 0 \leq b \leq df.$$

Therefore P_m is independent of $P_1, \dots, P_{m-2}$. Let x be the probability that $a_1 \dots a_h$ is examined. Then we have (assuming, without loss of generality, that h is odd)

$$x < P_G(a_1)G(a_2)P_G(a_3)G(a_4) \dots P_G(a_{h-2})G(a_{h-1})$$

and

$$x < P_G(a_2)G(a_3)P_G(a_4)G(a_5) \dots P_G(a_{h-1})G(a_h).$$

Thus

$$x < \sqrt{P_G(a_1)G(a_2)} \sqrt{P_G(a_2)G(a_3)} \dots \sqrt{P_G(a_{h-1})G(a_h)}$$

(for even or odd h).

Hence the expected number of terminal positions examined is less than

$$\begin{aligned} & \sum_{1 \leq a_1, \dots, a_h \leq df} \sqrt{P_G(a_1)G(a_2)} \sqrt{P_G(a_2)G(a_3)} \dots \sqrt{P_G(a_{h-1})G(a_h)} \\ &= f^h \sum_{1 \leq a_1, \dots, a_h \leq df} \sqrt{p_{a_1 a_2}} \sqrt{p_{a_2 a_3}} \dots \sqrt{p_{a_{h-1} a_h}} \\ &= f^h \sum_{1 \leq a_1 \leq df} \sum_{1 \leq a_2 \leq df} \sqrt{p_{a_1 a_2}} \sum_{1 \leq a_3 \leq df} \sqrt{p_{a_2 a_3}} \dots \sum_{1 \leq a_h \leq df} \sqrt{p_{a_{h-1} a_h}} \end{aligned}$$

which is $r^h C_{1,h}$, where the sequences $c_{i,n}$, $1 \leq i \leq d$, are defined by

$$\begin{aligned} c_{i,0} &= 1 \text{ for } 1 \leq i \leq d \\ (4.4) \quad c_{i,n+1} &= \sum_{1 \leq j \leq d} v p_{ij} c_{j,n}, \text{ for } 1 \leq i \leq d. \end{aligned}$$

Now define generating functions C_i , for $1 \leq i \leq d$, as follows:

$$C_i(z) = \sum_{n \geq 0} c_{i,n} z^n$$

Then (4.4) is equivalent to

$$C_i(z) - 1 = \sum_{1 \leq j \leq d} v p_{ij} z C_j(z) \text{ for } 1 \leq i \leq d.$$

Set $C(z) = (C_1(z) \dots C_d(z))^T$, and define the matrix

$$Z = \begin{pmatrix} z v p_{11} & z v p_{12} & \dots & z v p_{1d} \\ z v p_{21} & z v p_{22} & \dots & z v p_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ z v p_{d1} & z v p_{d2} & \dots & z v p_{dd} \end{pmatrix}.$$

Then $(-1 \dots -1)^T = (Z-I)C$, where I is the identity matrix. By Cramer's rule, $C_1(z) = U(z)/V(z)$, where U and V are polynomials defined by

$$U(z) = \det \begin{pmatrix} -1 & z v p_{12} & \dots & z v p_{1d} \\ \vdots & z v p_{22} & \dots & z v p_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ -1 & z v p_{d2} & \dots & z v p_{dd} - 1 \end{pmatrix}$$

and $V(z) = \det(z - I)$.

Note that r is an eigenvalue of $M_{d,f}$ if and only if $1/r$ is a root of $V(z)$. Since $C_1(z)$ is a quotient of polynomials,

it can be represented [40] as

$$C_1(z) = \sum_{1 \leq k \leq n} G_k(1/(z-B_k)),$$

where $B_1, \dots, B_n$ are the distinct roots of V , and $G_1, \dots, G_n$ are polynomials such that the degree of G_i is the multiplicity of B_i .

Every matrix of real, positive elements possesses one positive eigenvalue of multiplicity one that is strictly larger, in absolute value, than all the other eigen-

values [41]. $M_{d,f}$ is positive; let $r_1, i = 1, \dots, n$, be its eigenvalues, with r_1 the largest. If the eigenvalues $r_1 = 1/B_1, \dots, r_n = 1/B_n$ of $M_{d,f}$ are distinct, we have

$$\begin{aligned} C_1(z) &= E + \sum_{1 \leq i \leq d} e_i/(z-1/r_i) = E + \sum_{1 \leq i \leq d} -e_i r_i/(1-zr_i), \\ &= E + \sum_{n \geq 0} \sum_{1 \leq i \leq d} -e_i r_i^n z^n. \end{aligned}$$

Since r_1 is the largest of the r_i , $C_{1,h} = O(r_1^h)$. If the eigenvalues of $M_{d,f}$ are not distinct, the representation of $C_1(z)$ involves polynomials of degree higher than one. Even so, the linear term containing r_1 still dominates.

Q.E.D.

Lemma 4.4. Suppose the real-valued sequence $a_1, a_2, a_3, \dots$ obeys the rule

$$a_{m+n} \leq a_m + a_n \quad m, n = 1, 2, 3, \dots$$

Then the sequence $a_1/1, a_2/2, a_3/3, \dots$ either diverges to $-\infty$ or converges.

Proof. It suffices to consider the case where the $\liminf$, q , of the second sequence is finite. Let $\epsilon > 0$, and choose m such that $a_m/m < q + \epsilon$. Since every integer n can be expressed as $n = qm + r$ with $0 \leq r < m$, we have

$$a_n = a_{qm+r} \leq qa_m + a_r.$$

hence

$$\frac{a_n}{n} = \frac{a_{qm+r}}{qm+r} \leq \frac{qa_m + a_r}{qm+r} = \frac{a_m}{m} \frac{qm}{qm+r} + \frac{a_r}{qm+r} + \frac{a_r}{n}$$

hence

$$\frac{a_n}{n} < (q+\epsilon) \frac{qm}{qm+r} + \frac{a_r}{qm+r} + \frac{a_r}{n}$$

$$\text{hence } \limsup_{n \rightarrow +\infty} \frac{a_n}{n} = q$$

$$\text{and so } \lim_{n \rightarrow +\infty} \frac{a_n}{n} = q.$$

Q.E.D.

Definition. Let $T(d, h)$ be the number of terminal positions examined by a given algorithm in a lookahead tree of degree d and height h . The branching factor of T is

$$\lim_{h \rightarrow \infty} T(d, h)^{1/h},$$

if the limit exists.

Theorem 4.5. Let $T(d, f, h)$ be as defined in Theorem 4.4.

Then the branching factor of T ,

$$(4.5) \quad B = \lim_{h \rightarrow \infty} T(d, h, f)^{1/h},$$

satisfies

$$\frac{dfc_3}{\log df} \leq B \leq \frac{dfc_4}{\log d}$$

for certain constants $c_3, c_4 > 0$ independent of d and f .

Proof. Since $T(d, h_1, f)T(d, h_2, f)$ is the number of positions that would be examined by Pbound if "limit" were set to $+\infty$ for all positions at height h_1 in a lookahead tree of depth $h_1 + h_2$, we have $T(d, h_1 + h_2, f) \leq T(d, h_1, f)T(d, h_2, f)$. Hence by Lemma 4.4 applied to $\log T(d, h, f)$, the limit in (4.5) exists.

Lower bound: The parallel d - β routine without deep cutoffs, Pbound, examines at least as many nodes as its serial counterpart, bound, since each "limit" in the parallel case is greater than its counterpart in the serial case. As mentioned above, Knuth has proven that the branching factor of the number of terminal positions examined by bound in a tree of depth h and degree df is greater than or equal to $dfc_3/\log(df)$.

Upper bound: Let s and t be positive real numbers with $1/s + 1/t = 1$, and let E be an eigenvalue of the matrix $A = (a_{ij})$. Suppose $Ax = Ex$. Then

$$\begin{aligned} |E| \left(\sum_i |x_i^s| \right)^{1/s} &= \left(\sum_j \left| \sum_i a_{ij} x_i \right|^s \right)^{1/s} \\ &\leq \left(\sum_j \left(\sum_i |a_{ij}|^s \right)^{1/s} \left(\sum_i |x_i|^s \right)^{1/s} \right)^{1/s}, \end{aligned}$$

by Holder's inequality;

hence $|E| \leq \left(\sum_i \left(\sum_j |a_{ij}^t| \right)^{s/t} \right)^{1/s}$.

We will use this inequality to show that $r(d, f) \leq c_4 d / \log d$, for a certain constant c_4 and for $r(d, f)$ as defined in Theorem 4.4. Let $a_{ij} = \sqrt[p_{ij}]{}^t$, $E = r(d, f)$, and use approximation (4.3) for p_{ij} . For all s and t such that $1/s + 1/t = 1$, we have

$$\begin{aligned} r(d, f) &\leq \left(\sum_{1 \leq i \leq d} \left(\sum_{1 \leq j \leq d} ((i-1)f+1)^{-t((j-1)/2d)} \right)^{s/t} \right)^{1/s} \\ &\leq \left(\sum_{1 \leq i \leq d} \left(\sum_{1 \leq j \leq d} i^{-t((j-1)/2d)} \right)^{s/t} \right)^{1/s} \\ &\leq c_4 d / \ln d, \text{ for a suitable } s, t, \text{ and } c_4, \text{ by (4.1).} \end{aligned}$$

Theorem 4.4 and this upper bound for $r(d, f)$ give us the desired upper bound on the branching factor.

Q.E.D.

Theorem 4.5 deals with lookahead trees that are the same depth as the processor tree that searches them. In the next theorem we extend the analysis to the more general situation in which the lookahead tree can be deeper than the processor tree.

Theorem 4.6. The expected number of terminal positions examined by P_{bound} in a random uniform game tree of degree d and height $q+s$, evaluated by a processor tree of degree d and height q , where $d \geq 2$, $q \geq 0$ and $f \geq 1$, is asymptotically less than

$$c_5(d, f) f^q r(d, f)^q r_1(df)^s,$$

where $r(d, f)$ was given upper and lower bounds in Theorem 4.5, and r_1 satisfies

$$\frac{dfc_3}{\log(df)} \leq r_1(df) \leq \frac{dfc_4}{\log(df)}$$

for the constants c_3 and c_4 appearing in Theorem 4.5, and where $c_5(d, f)$ is a constant independent of q and s .

Proof. Since the values of the positions assigned for evaluation to leaf processors have random values, Theorem 4.4 implies that the number of these positions P satisfies

$$P < c(d, f) f^q r(d, f)^q.$$

Theorem 4.5 tells us that $r(d, f)$ satisfies

$$\frac{dc_3}{\log(df)} \leq r(d, f) \leq \frac{dc_4}{\log d}$$

If we set "limit" at level q of the lookahead tree to $+\infty$, then each leaf processor evaluating one position at level q would examine less than $c_2(df)r_1(df)^s$ terminal positions [38], where $r_1(df)$ satisfies

$$\frac{dfc_3}{\log(df)} \leq r_1(df) \leq \frac{dfc_4}{\log(df)}$$

and $c_2(df)$ is a constant independent of s .

The result follows with $c_5(d, f)$ set to $c(d, f)c_2(df)$.

Q.E.D.

4.7.5. Discussion of Theorem 4.6

In searching a lookahead tree of degree df and height $q + s$, the serial algorithm examines, on the average, at least

$$c_1 \left(\frac{dfc_3}{\log(df)} \right)^{q+s}$$

terminal nodes, where c_1 depends only on df and c_3 is a constant. The parallel algorithm examines less than

$$(4.6) \quad c_5 f^q \left(\frac{dc_4}{\log d} \right)^q \left(\frac{dfc_4}{\log(df)} \right)^s$$

terminal nodes on the average.

Under best-first and worst-first ordering, the finishing time for Palphabeta can be accurately estimated by dividing the amount of work to be done by the number of workers (terminal processors). This method of estimation is somewhat optimistic when applied to Pbound or the Tree-Splitting Algorithm under random ordering, because in Pbound a master waits until all successors in a batch of f have been evaluated before assigning the next batch, and in both Pbound and the Tree-Splitting Algorithm a master waits until the last successor is evaluated before receiving another position.

While we await more powerful methods, let us make the estimate anyway. Dividing (4.6) by the number of terminal processors, f^q , gives us

$$c_5 \left(\frac{dc_4}{\log d} \right)^q \left(\frac{dfc_4}{\log(df)} \right)^s$$

as the finishing time, and so the speedup would be at least

$$\frac{c_1}{c_5} \left(\frac{c_3}{c_4} \right)^{s+q} f^q \left(\frac{\log d}{\log(df)} \right)^q.$$

The factor $(c_3/c_4)^{s+q}$ appears in this expression because we used an optimistic bound for the serial algorithm and a pessimistic bound for the parallel algorithm. We can most likely remove it. The resulting expression is of order

$$\left(\frac{f \log d}{\log d + \log f} \right)^q$$

Recall that speedup under worst-first ordering is of order f^q ,

and by Theorem 4.3, speedup under best-first ordering is of order

$$f^{q/2}.$$

Speedup under both random and best-first ordering is clearly less than speedup under worst-first ordering. Speedup under random ordering is asymptotically greater than speedup under best-first ordering whenever

$$\frac{f \log d}{\log d + \log f} > \sqrt{f},$$

i.e. whenever

$$d > \frac{1}{\sqrt{f} - 1}$$

4.8. MANDATORY-WORK-FIRST SEARCH

Under best-first ordering of the lookahead tree, Palphabeta achieves only $O(k^{1/2})$ speedup with k processors. The cause of this inefficiency is clear: As Palphabeta evaluates group after group of children, if one of the children in the group is sufficiently good to produce a cutoff, then all of the work performed on its younger siblings within that group is wasted. If the tree is ordered best-first, all slaves but the first perform needless work. The serial algorithm, under the same ordering, avoids searching these younger siblings.

This section investigates an approach to avoid this extra work in a parallel alpha-beta algorithm. This approach exploits the "mandatory-work-first" distinction first proposed by Akl, Barnard and Doran [37]. By using this distinction to explicitly schedule node evaluations within a tree of processors, we produce a distributed algorithm whose finishing time can be calculated for a given regular processor tree and a given best-first or worst-first lookahead tree. This calculation will show that speedup obtained is "almost optimal" in the number of pro-

cessors used, in a sense that we will make clear later.

The algorithm, which we will call "mwf" as short for "mandatory-work-first", is a parallelization of the serial alpha-beta algorithm without deep cutoffs. We briefly review that algorithm, called bound:

```

function bound(p : position; limit : integer) : integer;
var m,i,t,d : integer;
begin
    determine the successors  $p_1, \dots, p_d$ ;
    if  $d = 0$  then return(staticvalue(p)) else
        begin m := -∞
            for i := 1 to d do
                begin t := -bound( $p_i, -m$ );
                    if  $t > m$  then  $m := t$ ;
                    if  $m \geq \text{limit}$  then return(m);
                end;
            end;
        return(m);
    end;
end;
```

Bound misses some cutoffs achieved by alphabeta, but not many. Under best-first ordering, the tree searched by bound(p, ∞) can be described as follows (Figure 4.3): All nodes searched are either type 1 or type 2. The root node is type 1. The first child of a type-1 node is type 1; the remaining children are type 2. The first child of a type-2 node is type 1; the remaining children are cut off. As Knuth and Moore show [38], the branching factor of the tree just described is

implementation and suggest modifications later. We will content ourselves with an algorithm that is accurate, if not quick, when moves are not ordered best-first. (If we only insist that the algorithm be accurate when moves are ordered best-first, we could cheat by always picking the first move.) In particular, *mwf* will simply use the routine intended for evaluation of type-1 nodes when re-evaluating type-2 nodes, even though more sophisticated re-evaluations are possible.

The algorithm uses three functions. "*Mwf1(p)*" is called on the root node and other type-1 nodes. The position *p* is to be evaluated. "*Mwf2(p)*" is called on type-2 nodes. The position *p* is to be partially evaluated. "*Mwf1(p)*" is called on positions *p* that need to be re-evaluated. "Alpha-beta" is the serial alpha-beta algorithm with deep cut-offs. It is used by terminal processors to evaluate nodes assigned to them. We assume a processor tree of height *q* and fanout *f*. The lookahead tree is of height *q*+*s* and degree *d*. The processor descriptors *l* through *d* denote the *d* slave processors. If *p* is a processor descriptor, then "*p.fn()*" denotes a remote call of the function "*fn*" on processor *p*. Here is the mandatory-work-first algorithm:

```

1 function mwfl(position p) : integer ;
2 var i,d : integer ;
3 t : array[1..f] of integer;
4 j : processor ;
5 begin
6   if I am a leaf processor then
7     return(alpha-beta(p,-∞,+∞));
8   determine the successors p1, ..., pd;
9   perfor i := 1 to d do
10     when a slave j is idle do
11       if i = 1 then t[i] := - j.mwfl(p1);
12       else t[i] := - j.mwf2(p1);
13   mwfl := t[1];
14   perfor i := 2 to d do { re-evaluate if needed }
15   begin
16     when a slave j is idle do
17       if t[i] > mwfl then
18         begin
19           t[i] := - j.mwfl(p1);
20         beginincr
21           if t[i] > mwfl then mwfl := t[i];
22         endincr;
23       end;
24   end;
25 end;

```

Several constructs in *mwfl* need explanation: First, the construct *perfor* (lines 9 to 12 and 14 to 24) denotes a parallel for-loop. Conceptually, a separate process is created for each iteration of the loop. After all of the processes have completed their iteration, the program continues as a single process at the next statement after the

parfor loop. Second, the construct "when <CONDITION> do <BODY>" (lines 10, 17) is similar to "await" in conditional critical regions [42]. The process pauses before <BODY>, proceeding only when <CONDITION> becomes true. This combination of blocking and parallel for-loop implements a queue of positions with the slaves as servers. Third, the construct "begincrit <STMT.LIST> endcrit" denotes a critical region. Only one process is allowed inside the critical region at a time. The use of a critical region (lines 20 to 22) ensures that the comparison and assignment of *mwfl* is an atomic operation. The function *mwfl* calls *mwf2* repeatedly (line 12). Here is *mwf2*:

```
function mwf2(p : position) : integer ;
p1 : position;
begin
    generate the first successor position p1;
    mwf2 := - mwfl(p1);
end;
```

4.8.1. Best-First Order

We now analyze the finishing time of *mwf* under best-first ordering of the lookahead tree. Define $a(i, j)$ to be the finishing time of *mwf* in evaluating a type-1 node of height j (in the lookahead tree) on a processor tree of height i and fanout f . An interior processor evaluates a type-1 node by assigning the node's children to its slaves. The $d-1$ type-2 children are partially evaluated on these slaves by evaluating their type-1 children. If performed

by one slave, these evaluations would therefore take time $a(i-1, j-1) + (d-1)a(i-1, j-2)$,

not counting message-passing time. With f slaves to do the work, the best time would cut this figure by a factor of f . The worst time occurs if with one type-2 position left in the queue, all f slaves finish their current task simultaneously, and only one can be assigned the final task.

The finishing times would be

$$\frac{a(i-1, j-1) + (d-1)a(i-1, j-2)}{f} + m$$

in the best case and

$$\frac{a(i-1, j-1) + (d+f-2)a(i-1, j-2)}{f} + m'$$

in the worst case, where m and m' denote message-passing times. Although m and m' depend on d and f , they are independent of i and j . At the terminal processors, a type-1 node is evaluated with serial alpha-beta search with deep cutoffs. Hence

$$a(0, j) = 2d^{j/2}.$$

To solve this two-dimensional recurrence relation, we need the following lemma. We omit the proof, which involves straightforward algebraic manipulations.

Lemma 4.5. Suppose the two-dimensional sequence $a(i, j)$ obeys the recurrence relation

$$a(i, j) = M(a(i-1, j-1) + Na(i-1, j-2)) + K,$$

and that

$$a(0, j) = Ad^{j/2},$$

where M, N, K , and A are positive real numbers. Then

$$a(i, j) = AM^i d^{(j-1)/2} (1 + Nd^{-1/2})^i + K \frac{(M+MN)^{i-1}}{M+MN-1}.$$

This lemma allows us to prove the following theorem.

Theorem 4.7. Suppose that mwf runs on a processor tree of depth $q \geq 1$ and fanout f . Suppose that the lookahead tree to be searched is arranged in best-first order and is of degree $d \geq 2$ and depth $q+s$, where $s \geq q$. Let X be the finishing time of mwf . Then

$$X \geq 2(1/f) q_d^{s/2} (1+(d-1)d^{-1/2})^q + m f \frac{(d/f)^{q-1}}{d-f}$$

and

$$X \leq 2(1/f) q_d^{s/2} (1+d^{1/2})^q + m' f \frac{(d/f)^{q-1}}{d-f}$$

Proof. For the first inequality, substitute

$$\begin{aligned} & 1/f \text{ for } M, \\ & d-1 \text{ for } N, \\ & m \text{ for } K, \\ & 2 \text{ for } A, \\ & q+s \text{ for } j, \text{ and} \\ & q \text{ for } i \end{aligned}$$

in the formula given by Lemma 4.5 for $a(i, j)$. For the second inequality, substitute

$$\begin{aligned} & 1/f \text{ for } M, \\ & d+f-2 \text{ for } N, \\ & m' \text{ for } K, \\ & 2 \text{ for } A, \\ & q+s \text{ for } j, \text{ and} \\ & q \text{ for } i \end{aligned}$$

in the formula given by Lemma 4.5 for $a(i, j)$.

Q.E.D.

Now that we have calculated the finishing time of the parallel algorithm, we can express the speedup in terms of the number of terminal processors.

Corollary 4.2. Under the assumptions of Theorem 4.7, and under the additional assumption that $s \gg q$, the speedup S of the mwf algorithm with P terminal processors satisfies

$$\frac{1 - \ln_f(1+d^{-1/2})}{P} + \frac{1 - \ln_f(f-2)d^{-1}}{(f-2)d^{-1}} \leq S \leq P$$

Proof. For the lookahead tree under consideration, Corollary 4.1 (Section 4.7.2) says that the finishing time for the serial alpha-beta algorithm with deep cutoffs is approximately

$$2d(s+q)/2.$$

If we divide this quantity by the bounds given on the finishing time given by Theorem 4.7 (Section 4.8.1), and take the limit as s goes to ∞ , we obtain the desired result.

Q.E.D.

As d increases, the speedup approaches P , the number of terminal processors. Hence we use the term "almost optimal" to describe mwf under best-first ordering.

4.8.2. Worst-First Order

We defined worst-first order earlier as a particularly poor ordering of the tree under which the serial alpha-beta algorithm achieves no cutoffs. We now analyze the finish-

ing time of $mwfl$ under the assumption that the lookahead tree is arranged in worst-first order. $mwfl$ running on a processor P evaluates a node N by queuing N 's children for evaluation on P 's slaves. N 's first child is evaluated with $mwfl$, and the others are partially evaluated with $mwf2$. When partial evaluations are finished, $mwfl$ discovers it must re-evaluate each of N 's $d-1$ younger children. These children are queued for re-evaluation by $mwfl$ on P 's slaves. In all, we have d invocations of $mwfl$ and $d-1$ invocations of $mwf2$. As in the best-first ordering of the lookahead tree, the f slaves can finish most quickly by finishing their last assignments simultaneously, or most slowly by finishing simultaneously with one more task to be performed. The finishing times would be

$$\frac{da(i-1, j-1)}{f} + \frac{(d-1)a(i-1, j-2)}{f} + m$$

in the best case and

$$\frac{(d+f-1)a(i-1, j-1)}{f} + \frac{(d-1)a(i-1, j-2)}{f} + m'$$

in the worst case. A terminal processor evaluates its assigned nodes with the serial alphabet algorithm. We assume that the serial algorithm evaluates a tree in time equal to the number of leaf nodes on the tree. Hence $a(0, j) = d^j$. With these recursive relationships, we can use Lemma 4.5 to calculate bounds on the finishing times and speedups for $mwfl$ under worst-first ordering.

Theorem 4.8. Suppose that $mwfl$ runs on a processor tree of

depth $q \geq 1$ and fanout f . Suppose that the lookahead tree to be searched is arranged in worst-first order and is of degree $d \geq 2$ and depth $q+s$, where $s \geq q$. Denote by x the finishing time of $mwfl$. Then

$$x \geq (d/f)q_d^q (1 + (d-1)/d^2)^q + m' \frac{(d/f)q-1}{d-f}$$

and

$$x \leq ((d+f-1)/f)q_d^q (1 + (d-1)/((d+f-1)d))^q + m' \frac{(d/f)q-1}{d-f}$$

Proof. For the first inequality, substitute

$$\begin{aligned} & d/f \text{ for } M, \\ & (d-1)/d \text{ for } N, \\ & m \text{ for } K, \\ & 1 \text{ for } A, \\ & d^2 \text{ for } d, \\ & q+s \text{ for } j, \text{ and} \\ & q \text{ for } i \end{aligned}$$

in the formula given by Lemma 4.5 for $a(i, j)$. For the second inequality, substitute

$$\begin{aligned} & (d+f-1)/f \text{ for } M, \\ & (d-1)/(d+f-1) \text{ for } N, \\ & m' \text{ for } K, \\ & 1 \text{ for } A, \\ & d^2 \text{ for } d, \\ & q+s \text{ for } j, \text{ and} \\ & q \text{ for } i \end{aligned}$$

in the formula given by Lemma 4.5 for $a(i, j)$.

Q.E.D.

Corollary 4.3. Under the assumptions of Theorem 4.8, and under the additional assumption that $s \gg q$, the speedup S of the $mwfl$ algorithm with P terminal processors satisfies

$$\frac{1 - \ln_f(1 + d^{-1} - d^{-2})}{p} \leq s \leq p \frac{1 - \ln_f(1 + d^{-1} - d^{-2})}{p}$$

Proof. For the lookahead tree under consideration, the finishing time for the serial alpha-beta algorithm with deep cutoffs is the number of terminal nodes, which is d^{s+q} .

If we divide this quantity by the bounds given on the finishing time given by Theorem 4.8, and take the limit as s goes to ∞ , we obtain the desired result.

Q.E.D.

4.8.3. Other Orderings

Mwf is deficient in ways that the analysis above does not reveal. First, the re-evaluation of a partially evaluated node searches all the node's children, even though the first child has already been evaluated. A more sophisticated algorithm would resume the search at the second child. This deficiency does not appear under best-first order because no nodes are re-evaluated, and does not significantly affect the algorithm's performance under worst-first order because the re-evaluation involves d times as much work as the partial evaluation. Second, mwf does not attempt to pass $q-p$ values to recursive calls on itself, even when these windows are available. This deficiency is insignificant under best-first order because mwf "predicts" all shallow cutoffs that such values could pro-

duce. Under worst-first order, cutoffs are not possible and so windows are useless.

These deficiencies occur in the murky area between best-first and worst-first ordering of uniform lookahead trees. The only other "benchmark" lookahead tree available for theoretical treatment is the tree of uniform depth and height with randomly distributed terminal values. But as we have already seen, analyses assuming random ordering are fairly difficult. Further research in this area might be directed toward creating other analyzable orderings of lookahead trees. For example, one might consider a lookahead tree that is originally randomly ordered. As it is being searched, however, heuristics are applied that successfully reorder best branches first in a certain percentage of cases. By making this percentage a parameter, an analysis might be able to model practical situations.

Mwf is a parallelization of the serial algorithm without deep cutoffs. A parallelization of the serial algorithm with deep cutoffs might be possible. Such a parallel algorithm would likely be more complicated than mwf, but might be more efficient.

4.9. COMPARISON OF PALPHABETA AND MWF

We have now analyzed two different parallel alpha-beta algorithms, Palphabeta and mwf, under conditions of best-first and worst-first ordering. In each of the four possible (algorithm, ordering) combinations, we have derived a formula representing the speedup gained with P terminal processors. Table 4 summarizes these formulas.

Ordering of Lookahead Tree

	$\frac{\text{Best-First}}{P/2}$	$\frac{\text{Worst-First}}{P}$
Palphabeta:		
mwf:		
upper bound	$P^{1-\ln_f(1+d^{-1/2}-d^{-1})}$	$P^{1-\ln_f(1+d^{-1}-d^{-2})}$
lower bound	$P^{1-\ln_f(1+d^{-1/2}+(f-2)d^{-1})}$	$P^{1-\ln_f(1+fd^{-1}-d^{-2})}$

Table 4. Speedup in Parallel Alpha-Beta Search

The speedups for mwf depend on d , the degree of the lookahead tree, and f , the fanout of the processor tree. It is instructive to substitute actual values for d and f . For example, the average number of moves from a position in the game of chess is about 38. For a best-first lookahead tree of degree 38 and processor tree of fanout 2, Corollary 4.2 predicts that speedup for mwf will satisfy

$$P^{0.78} \leq S \leq P^{0.82},$$

which is significantly better than Palphabeta. For a worst-first lookahead tree of degree 38 and processor tree of fanout 2, Corollary 4.3 predicts that speedup for mwf will satisfy

$$P^{0.93} \leq S \leq P^{0.96},$$

which is almost as good as Palphabeta.

4.10. TIPS FOR PROCESSOR-TREE ARCHITECTS

Anyone designing hardware to play a game like chess faces a number of decisions along the way. Our discussion raises the following questions:

1. Should parallel processing be used?
2. If so, how powerful should the leaf processors be?
3. How many leaf processors should be used?

Since our algorithms all reduce to the serial algorithm when $q=0$, the third question is really a generalization of the first. To help answer these questions, we make the following simplifying assumptions:

1. A certain fixed amount of money may be spent.
2. The parallel algorithm gives P^e speedup with P processors for some fixed $0 < e \leq 1$.
3. Several different serial processors are available.

Associated with each of these processors is a dollar cost and a processing speed in units of positions ex-

amined per second. These data are described by a "serial power function", $W(S)$, that tells how much power W we can obtain in a serial processor by spending S dollars. We will assume that $W(S)$ is continuously differentiable.

4.10.1. Serial versus Parallel

We will start with a concrete example. Suppose that for \$40,000 we can buy a processor that searches 250 nodes per second, and for \$10,000 we can buy a processor that searches 100 nodes per second. Suppose that we have \$40,000 to spend and a parallel algorithm that gives $p^{0.5}$ speedup with P processors. We can spend our money to buy one \$40,000 processor or four \$10,000 processors. We know that our algorithm will give $4^{0.5}$ speedup, or 200 nodes per second with the four processors. Hence we would be wise to buy the \$40,000 processor and use the serial algorithm to evaluate 250 instead of 200 nodes per second. In general, when we multiply the amount of money available to us by any constant k , we do better to use serial processing if we gain more than k^e speedup with the more expensive serial processor.

We define the critical point, if it exists, to be that number of dollars S such that optimal systems less expensive than S are serial machines, and optimal systems more

expensive than S are parallel machines. If $W(S)$ follows the classic economic pattern of diminishing returns on investment for large S (Figure 4.4), then the critical point occurs when

$$\frac{d \log W}{d \log S} = e.$$

The following theorem helps to formalize this result.

Theorem 4.9. Suppose that W is a positive, differentiable function defined on the positive real numbers. Suppose that $e > 0$, and that there exists a positive number S_0 such that for $S \geq S_0$,

$$\frac{d \log W}{d \log S} \leq e.$$

Then for $k > 1$,

$$k^e W(S_0) \geq W(kS_0).$$

Proof. The proof is by contradiction. Suppose that

$$k^e W(S_0) < W(kS_0).$$

Taking the log of both sides, we can rewrite this as

$$(4.7) \quad e < \frac{\log W(kS_0) - \log W(S_0)}{\log kS_0 - \log S_0}.$$

By the Mean Value Theorem, the right hand side of 4.7 is equal to the derivative

$$\frac{d \log W}{d \log S}$$

evaluated at some point S_1 such that $S_0 \leq S_1 \leq kS_0$. Hence

$$e < \frac{d \log W}{d \log S}.$$

at S_1 , which contradicts our original assumption.

Q.E.D.

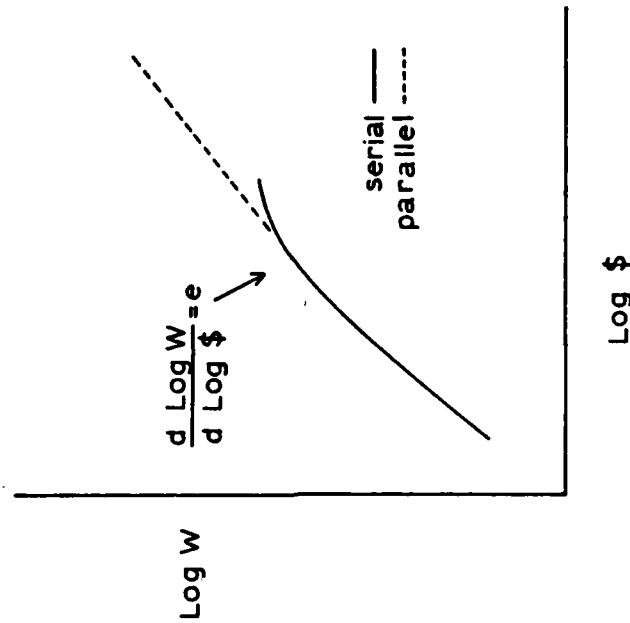


Fig. 4.4. Serial vs. Parallel Machines.

Theorem 4.9 says that if the marginal return on investment in serial machines after S_0 dollars is less than e , improvement in speed for every one percent increase in cost, then more speed can be obtained from k processors at S_0 dollars apiece than from a single machine at kS_0 dollars.

4.10.2. Maximal Processor Trees

Our analyses all assume that each message to a leaf processor invokes a substantial amount of work. For the moment, let us assume that a "substantial amount of work" consists of a search of a subtree of height one or more. As we have pointed out, when move selection must be completed with a certain time limit, we cannot gain unlimited speedup by adding more and more layers to the processor tree. Each additional layer in the processor tree buys less than one additional layer in the lookahead tree that can be searched within the time limit. Hence the subtrees assigned to terminal processors grow shorter and shorter, and eventually we violate our assumption about giving them substantial amounts of work per message. In this section we estimate how many processors are required before the assumption becomes false.

Suppose that a serial processor can search a lookahead tree L of depth D within the required time limit. Assume

that the branching factor (the ratio of the sizes of lookahead trees of successive depths when searched by the serial algorithm) is B , and that the fanout of the processor tree is f . Assume that the parallel algorithm we are using gives k^e speedup with k leaf processors, for some fixed $0 < e \leq 1$. A processor tree of height q gives f^{eq} speedup. Within the time limit it can therefore search a pruned lookahead tree that is f^{eq} times as big as L , or $\log_B f^{eq}$ ply deeper. Hence Palphabeta assigns subtrees of depth one to leaf processors when

$$D + \log_B f^{eq} = q + 1,$$

or when

$$(4.8) \quad q = \frac{D-1}{1-e \log_B f}.$$

When the processor tree is shorter than the q given by Equation 4.8, our advertised speedups can be met within the time limit. Hence for $p < f^q$ leaf processors, we obtain speedup p^e . Mwf assigns subtrees of height one to leaf processors when the height of the processor tree is one less than half the height of the lookahead tree. Hence for mwf, the equation corresponding to (4.8) is

$$q = \frac{D-1}{2-e \log_B f}.$$

Chapter 5 - Piecewise-Serial Iterative Methods

Beyond the Cray 2, a yet faster computer is taking shape in Mr. Cray's mind. "I do tend to look forward in my thinking and I don't like to rest on my laurels," he says. How fast could such a computer be? Perhaps, he says, a trillion calculations a second.

That prospect is an intriguing one for scientists. Says Sidney Fernbach, a scientific administrator at Livermore, "There's no machine that Seymour Cray can conceive that would be too fast for us."

- Wall Street Journal
(12 April 1979)

5.1. INTRODUCTION

Many numerical computations are locally defined and iterative: A rectangular array of numbers A_0 is given; $A_1, A_2, \dots$ are iteratively defined by a locally defined rule. That is, the value of an element in A_n is some function of the values of its immediate neighbors in A_{n-1} .

This chapter investigates locally-defined iterative computations on Arschne-like architectures. As a focus for our investigation we will consider the numerical solution of an important problem in engineering and physics, the Dirichlet problem.

In Section 2 we discuss the numerical solution of the Dirichlet problem. Section 3 discusses previous work in parallel iterative methods. Section 4 proposes a family of distributed algorithms for the iterative solution of the Dirichlet problem.

5.2. THE DIRICHLET PROBLEM

Let R be the interior and S the boundary of the unit square $0 \leq x \leq 1$, $0 \leq y \leq 1$. Let $g(x,y)$ be a continuous function defined on S . In the Dirichlet problem we seek a function $u(x,y)$ defined on $R + S$ that is twice continuously differentiable on R and satisfies Laplace's equation

$$(5.1) \quad u_{xx} + u_{yy} = 0$$

on R , and equals $g(x,y)$ on S . To approximate $u(x,y)$ we superimpose over $R + S$ a uniform mesh of $N+1$ horizontal and $N+1$ vertical lines with spacing $h = 1/N$, for some positive integer N . We call the $(N+1)^2$ intersections of these lines mesh points. To approximate u at a given internal mesh point (x,y) , we use the approximations

$$u_{xx} \approx [u(x+h,y) + u(x-h,y) - 2u(x,y)]/h^2$$

$$u_{yy} \approx [u(x,y+h) + u(x,y-h) - 2u(x,y)]/h^2$$

to rewrite (5.1) as

$$(5.2) \quad 4u(x,y) - u(x+h,y) - u(x-h,y) - u(x,y+h) - u(x,y-h) = 0.$$

This equation applied at interior points together with the

boundary condition

$$u(x,y) = g(x,y)$$

forms a discrete version of the Dirichlet problem.

5.2.1. Jacobi Method

Equation (5.2) specifies a set of $(N-1)^2$ linear equations in $(N-1)^2$ unknowns. This set of equations could be solved directly, but the sparsity of the matrix often makes iterative methods more efficient. We first solve equation (5.2) for $u(x,y)$, giving

$$u(x,y) = [u(x+h,y) + u(x-h,y) + u(x,y+h) + u(x,y-h)]/4.$$

Given "old" values $u_n(x,y)$ at mesh points, we use the following equation to generate "new" values $u_{n+1}(x,y)$:

$$(5.3) \quad u_{n+1} = [u_n(x+h,y) + u_n(x-h,y) + u_n(x,y+h) + u_n(x,y-h)]/4.$$

Equation (5.3) is applied iteratively until further iterations do not change u very much. This method is the Jacobi (J) method.

The Jacobi method is very slow for large N . Indeed, it is well known [43] that the number of iterations required for the Jacobi method to converge is proportional to N^2 . The total work needed is proportional to N^4 , since each iteration treats $O(N^2)$ internal mesh points.

Although slow, the Jacobi method is useful for two reasons. First, for many problems its intermediate

iterates correspond to the transient behavior of the physical process being modeled. Many other methods converge more quickly to the steady state, but do so by mathematical shortcuts not taken by the physical process being modeled. When we are interested in transient behavior for problems of heat flow, we must use methods similar to the Jacobi method. Second, some optimizations of J such as SOR (defined below) are unstable for some problems other than the Dirichlet problem.

5.2.2. Gauss-Seidel Method

The Gauss-Seidel (GS) method differs from the Jacobi method by using new neighbor values whenever available. For example, if the outer loop of each iteration visits rows from $y=0$ to $y=1$, and the inner loop visits mesh points in a row from $x=0$ to $x=1$, then GS calculates new values according to the formula

$$u_{n+1} = [u_n(x+h, y) + u_{n+1}(x-h, y) + u_n(x, y+h) + u_{n+1}(x, y-h)]/4.$$

The GS method needs only half as many iterations to converge as the J method. This speedup, though significant, is independent of N . Hence GS also needs $O(N^2)$ iterations to achieve convergence.

5.2.3. Successive Over-Relaxation

GS optimizes J by using new values whenever available. Successive Over-Relaxation (SOR) optimizes GS by "over-correcting" from one iteration to the next. If GS computes the value u'_{n+1} by adding the increment $u'_{n+1} - u_n$ to u_n , then SOR computes u_{n+1} by adding an even greater increment:

$$u_{n+1} = u_n + w(u'_{n+1} - u_n).$$

The relaxation parameter w is usually between 1 and 2; if 1, then SOR reduces to GS. Much work has been done to determine optimum values of w . For the Dirichlet Problem on the unit square with mesh spacing h , it can be shown that the optimum value of w is $2/(1 + \sin(\pi h))$ [43]. For example, if h is $1/20$ then the optimal value for w is 1.72945.

With an optimal value of w , SOR requires $O(N)$ iterations to converge.

5.3. PREVIOUS WORK

In this section we review parallel algorithms that have been developed for locally defined iterative methods.

Stone [28] notes that many serial techniques are not directly applicable in parallel algorithms. For example, the serial Gauss-Seidel technique uses newly-computed values for neighboring points wherever possible. If new

values for all points are calculated simultaneously, then Gauss-Seidel cannot be used.

Rosenfeld [44] simulates the operation of a C.mmp-like machine (that is, all processors have equal access to all memories) in the computation of the distribution of current in an electrical network. He shows that with proper programming aimed at reducing storage interference, N processors can give nearly N -fold speedup when N is less than about 10.

Weiman [45] proposes an L by M grid of microprocessors to perform iterative calculations on an L by M by N -point mesh arising from the Navier-Stokes equation. Processors are connected in the four compass directions. For a certain range of problem size, each cell needs approximately 2K words of storage, a small number of registers, and a small processor. Cell (i, j) holds all data points with spatial coordinates (i, j, x) ; after each time step it communicates all newly-computed values to all four neighbors.

Flanders [6] has built a 32-by-32 SIMD array of one-bit microprocessors called the Distributed Array Processor (DAP). Communications lines connect each processor to its neighbors in the four cardinal directions. Finite-difference calculations are performed by mapping the processors one-to-one onto the points of the problem grid. If the problem grid is larger than the processor grid, then

the calculation for each time step must successively load the processor array with "patches" from the problem grid. Flanders estimates that a 64-by-64 DAP array would perform finite-difference calculations at a rate 20 times that of an IBM 360/195.

Weich [46] reports measurements of calculations used in atmospheric simulation models on the Pepe Parallel Processor, which is an SIMD machine with data transfers on a shared bus. Measurements were taken on Pepe hardware with 11 processing elements (PEs). Extrapolation of these measurements indicates that a 161-PE Pepe would execute the Geophysical Fluid Dynamics Laboratory benchmark about 7 times faster than an IBM 360/195.

Parallel iterative methods usually perform exactly the same computation as some well-known serial method. An interesting exception is the chaotic relaxation technique of Chazan and Miranker [47] and Baudet [48]. Chaotic relaxation is an attempt to avoid time-consuming synchronization among multiprocessors performing iterative methods. For example, suppose several processors access a common memory to perform Jacobi's method. If we insist that exactly the same computation be performed as in the serial case, then when processor x is computing the value of a point whose neighbor p is computed by processor y , the two processors must synchronize their actions so that processor y 's compu-

tation of p for the n th iteration is stored before processor x accesses p . The overhead of synchronization can significantly slow down the computation. Chaotic relaxation doesn't bother with synchronization; x gets either the old or the new value for p . Theoretical analyses indicate that as long as one processor does not lag too far behind its neighbors, convergence is still assured, if slowed. Baudet's measurements on C.mmp show that the method pays off: The extra iterations needed for convergence are more than offset by the time saved by not synchronizing.

5.4. PIECEWISE-SERIAL ITERATIVE METHODS

This section analyzes several parallel algorithms for implementing the Jacobi method on a distributed system. As usual, we define speedup to be time for the serial algorithm divided by time for the parallel algorithm. The efficiency of a parallel algorithm is its speedup divided by the number of processors used.

5.4.1. Uniform Regions With Grid Topology

One natural way to solve the Dirichlet problem on a multicomputer architecture is the following: Arrange the processors in a q by q grid, with each processor connected to its nearest neighbors in the four compass directions.

We assume that the problem grid consists of pq by pq points. This grid is broken into q^2 square regions of size p^2 (Figure 5.1). If we index both the processor grid and problem grid by Cartesian coordinates, then processor (i,j) , for $0 \leq i,j < q$, contains those problem points (x,y) such that $iq \leq x < (i+1)q$ and $jq \leq y < (j+1)q$. Hence regions sharing a common border are assigned to neighboring processors.

We will call the following algorithm the grid algorithm. In order to compute values for the next time step according to the Jacobi method, each processor needs to know the current value of points bordering on its region. Prior to the computation of each time step, each processor communicates to each of its nearest neighbors values of its border points adjacent to that neighbor. We assume that a processor can send n numbers to a neighbor in time $p + n\delta$ (p is the per-message overhead, and δ is the time to send one number) and can compute one mesh point value in one unit of time. Each processor must send and receive p numbers to/from each of its four neighbors, so the communication phase of the grid algorithm takes

$$8p + 8p\delta$$

units of time. The computation phase takes p^2 units of time to compute p^2 points on each processor. We have proved the following result:

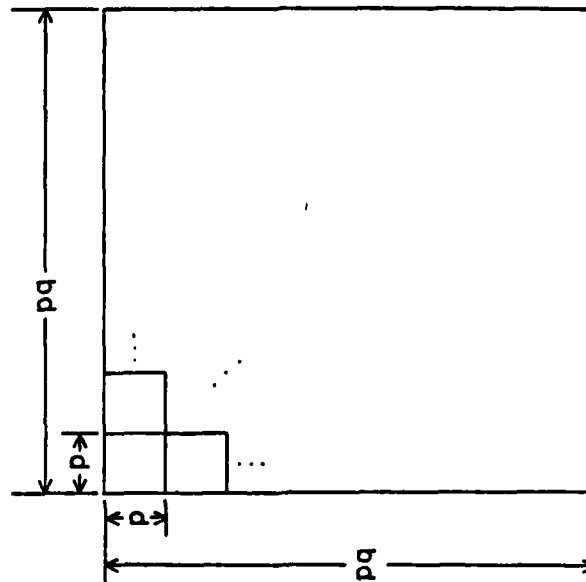


Fig. 5.1. Partition of problem grid for grid topology.

Theorem 5.1. The grid algorithm computes one iteration of the Jacobi method in

$$(5.4) \quad 8p + 8pq + p^2$$

units of time.

The first two terms of (5.4) represent message-passing time; the last term represents computation time. Speedup is

$$\frac{p^2 q^2}{8p + 8pq + p^2}$$

and efficiency is

$$\frac{p^2}{8p + 8pq + p^2}.$$

5.4.2. Uniform Regions With Tree Topology

A more flexible way to implement the Jacobi method on a multicomputer architecture is the Synchronous Tree Algorithm: Suppose that the problem grid is pn^q by pn^q , containing $p^2 n^{2q}$ points in all. Arrange the processors into a tree of height q and fanout n^2 . (The height of a tree with one node is zero.) Any non-terminal processor is called a master and its children are called slaves.

The root processor is responsible for the entire problem grid. Having n^2 slaves, it divides the grid into n^2 square sub-regions of size pn^{q-1} by pn^{q-1} and assigns each sub-region to a slave. Each of these n^2 slaves likewise divides its region into n^2 regions, assigning each to one

of its slaves, and so on. Terminal slaves receive a square region with p^2 mesh points (Figure 5.2).

Before each time step, any slave (terminal or otherwise) needs to know point values from the previous time step that border on its region. Its master provides these values. At the end of each time step, each slave sends to its master all of its border point values. The master will relay appropriate sets of values to its slaves before the next time step.

We now calculate the finishing time for one time step (Figure 5.3). Define a_k to be the time for a processor at distance k from the leaves of the processor tree to complete a time step. We start timing after the processor has gotten its border values from its master. We stop timing when the processor is ready to send border values to its master. Since a slave processor must calculate p^2 values, $a_0 = p^2$. A processor at distance $k+1$ from the leaves must first supply $4pn^k$ border values to each of its n^2 slaves. Each slave then takes time a_k and sends approximately $4pn^k$ border values ($4pn^k - 4$, to be exact) back to the master. At time $t=0$, we start timing. At time $t = i(p + 46pn^k)$ the i th slave, for $i = 1, \dots, n^2$, finishes receiving border points and starts computing values for the next iteration. At time $t = i(p + 46pn^k) + a_k$ the i th slave finishes its computation phase and starts to send border points to its

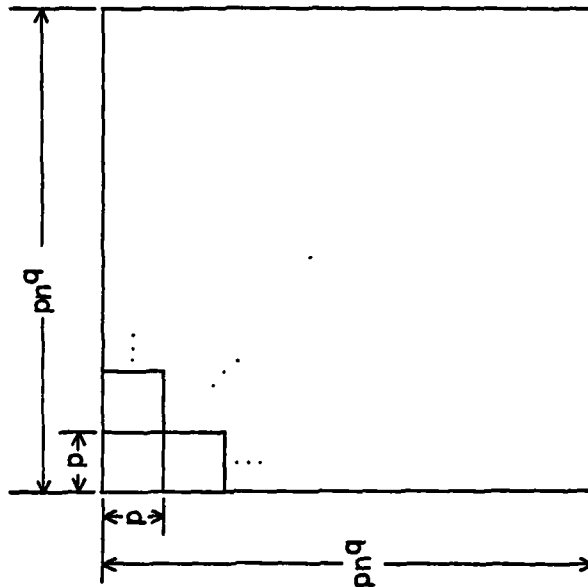


Fig. 5.2. Partition of problem grid for tree topology.

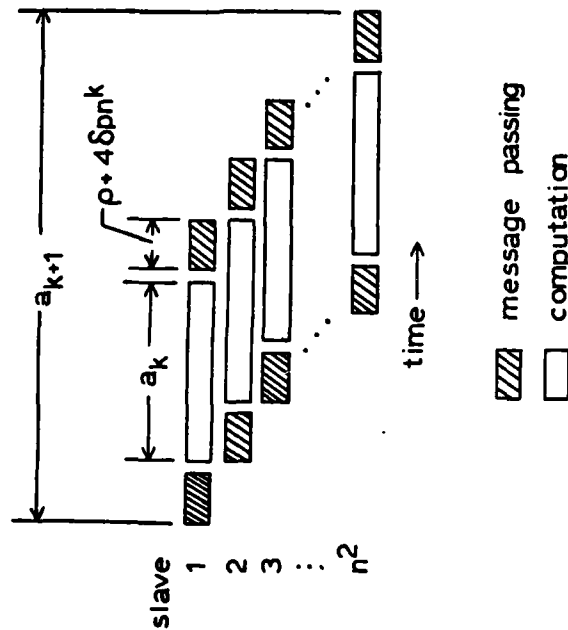


Fig. 5.3. One time step in the life of a master.

master. (The master may still be busy sending points to the latter slaves when the first slave wants to send points back to it. For the time being, we assume that if necessary a master can simultaneously send and receive without being slowed down in either activity. Later we will find conditions under which no overlap occurs.) At time $t = n^2(p + 4\delta pn^k) + a_k + p + 4\delta pn^k$, the last slave finishes sending border points to its master. We therefore have the recurrence relation

$$(5.5) \quad a_{k+1} = (n^2+1)(p + 4\delta pn^k) + a_k,$$

which is the finishing time in the synchronous algorithm for a master at height $k+1$, in terms of the finishing time of one of its slaves. The solution to this recurrence relation for $k = q$ is

$$a_q = (n^2+1)(qp + 4\delta p(n^q-1)/(n-1)) + p^2.$$

We have proved

Theorem 5.2. The finishing time a_q for the synchronous tree algorithm is

$$(5.6) \quad a_q = (n^2+1)(qp + 4\delta p(n^q-1)/(n-1)) + p^2.$$

Hence speedup is

$$\frac{p^2 n^2 q}{(n^2+1)(qp + 4\delta p(n^q-1)/(n-1)) + p^2}$$

and efficiency is

$$\frac{p^2}{(n^2+1)(qp + 4\delta p(n^q-1)/(n-1)) + p^2},$$

which is less than the efficiency of the grid algorithm.

Optimal Fan-out

We have assumed that the fanout of the processor tree is some perfect square n^2 . We now show that the optimal value of n is two. Suppose we have a fixed number of leaf processors C^2 . There may be several different pairs of integers (n, q) such that a tree of height q and fanout n^2 has $C^2 = n^{2q}$ terminal processors. As (5.6) shows, for a given problem the computation time for these several trees is identical (p^2), but the message-passing time may vary. We wish to find the optimal fanout n^2 ; i.e. we want to know which value $2 \leq n \leq C$ minimizes the message-passing time

$$(n^2+1)(qp + 46p(n^q-1)/(n-1)).$$

Theorem 5.3. Let p and C be arbitrary positive integers, and let p and 6 be positive real numbers. If we let n range over the integers greater than one, and require that $n^q = C$, then

$$(n^2+1)(qp + 46p(n^q-1)/(n-1)).$$

achieves its minimum value at $n = 2$.

Proof. We show that both

$$(n^2+1)q \quad \text{and} \quad (n^2+1)(n^q-1)/(n-1)$$

achieve their minimum value at $n = 2$.

First, $n^q = C$ and $q = (\ln C)/\ln n$. To minimize

$$(n^2+1)q = (n^2+1)(\ln C)/\ln n,$$

it would suffice to minimize $g(n) = (n^2+1)/\ln n$, since $\ln C > 0$. Since the derivative

$$g'(n) = \frac{2n(\ln n) - (n^2+1)/n}{\ln^2 n},$$

is always positive for $n \geq 2$, $g(n)$ achieves its minimum at $n = 2$.

To minimize

$$\frac{(n^2+1)(n^q-1)}{n-1},$$

we first note that $n^{q-1} = C-1 > 0$, so it would suffice to minimize

$$h(n) = \frac{n^2+1}{n-1}.$$

Again, the derivative $h'(n)$ is positive for $n \geq 3$, and $h(2) = h(3) = 5$, so h achieves its minimum at $n = 2$.

Q.E.D.

We have shown that 2^2 is the optimal perfect square fanout of the processor tree. This fact might suggest to us that a fanout of 2 might yield even greater efficiency. We now show that this conjecture is not true. Suppose, for example, that we have a tree of height $2q$ and fanout 2, yielding the same number of leaf processors, 2^{2q} , as a tree of height q and fanout 4. The master M at height $2k+2$ divides the $p^{2^{k+1}}$ by $p^{2^{k+1}}$ problem grid into two rectangular sub-regions, each of shape $p^{2^{k+1}}$ by p^{2^k} , and assigns each to one of its two slaves, s_1 and s_2 . Slave s_n divides the rectangular region into two square regions of side p^{2^k} , and assigns each to one of its slaves, s_{n1} and s_{n2} . Define b_k to be the time for a processor at height $2k$ to finish

one time step. We define a_q according to equation (5.5) with $n=2$.

Theorem 5.4. For a given number of leaf processors, the synchronous tree algorithm finishes sooner on a processor tree of fanout four than on a processor tree of fanout two. Proof. We develop a recurrence relation for b_k and compare it to the recurrence for a_k . We start our timing when M starts sending border points to s_1 . We have the following sequence of events: At $t=0$, M starts sending $3p2^{k+1}$ points to s_1 . At $t=p+3p2^{k+1}$, M starts sending $3p2^{k+1}$ points to s_2 . At $t=2p+6p2^{k+1}$, s_2 has received all points and starts sending $2p2^{k+1}$ points to s_{21} . At $t=3p+8p2^{k+1}$, s_2 starts sending $2p2^{k+1}$ points to s_{22} . At $t=4p+10p2^{k+1}$, s_{22} finishes receiving points and starts its computation. At $t=4p+10p2^{k+1}+b_k$, s_{22} finishes its computation and starts sending $2p2^{k+1}$ (minus 4, which we ignore) points back to s_2 . At $t=5p+12p2^{k+1}+b_k$, s_2 finishes receiving points from s_{22} and starts sending $3p2^{k+1}$ (minus 4) points back to M . At $t=6p+15p2^{k+1}+b_k$, M finishes receiving points from s_2 and is thus finished with its computation. Hence

$$b_{k+1} = 6(p + 5p2^k) + b_k,$$

whereas

$$a_{k+1} = 5(p + 4p2^k) + a_k.$$

Since $a_0 = b_0 = p^2$, we have $b_q > a_q$ for $q > 0$.

Q.E.D.

Among processor trees, we may now restrict our attention to those whose fanout is four. Equation (5.6), the finishing time for the tree architecture, simplifies to

$$(5.7) \quad a_q = 5qp + 206p(2^q - 1) + p^2,$$

which is the finishing time for a processor tree of height q and fanout four executing the synchronous algorithm.

Non-overlap of Send and Receive

We now return to the possibility that a slave might wish to send border points to its master before that master has finished giving border points to all the other slaves. Suppose that the master is at height q from the leaves of the processor tree. After the first slave has finished receiving its border points, two activities proceed in parallel (Figure 5.4). First, the first slave performs its computation. This activity takes time

$$(5.8) \quad 5(qp + 46p(2^q - 1)) + p^2,$$

according to Equation (5.7). Meanwhile, the master is sending border points to the other three slaves. This activity takes time

$$(5.9) \quad 3(p + 46p2^q).$$

If the computation (5.8) takes longer than the communication (5.9), we can be sure that the master will not need to receive results while it is sending values. We therefore require the condition

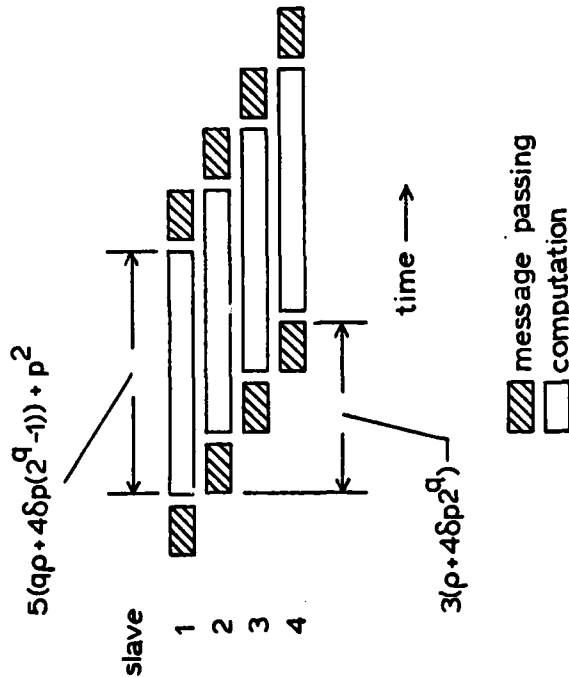


Fig. 5.4. Non-overlap of send and receive.

$$5(qp + 48p(2^q - 1)) + p^2 - 3(p + 48p2^q) > 0.$$

We can rewrite this inequality as

$$p(5q-3) + 6p(2^{q+3} - 20) + p^2 > 0.$$

The left-hand side increases monotonically with q ; hence we have the greatest difficulty in satisfying the inequality when $q=0$. It therefore suffices to show that

$$p^2 - 126p - 3p > 0.$$

By the quadratic formula, we have proved

Theorem 5.5. In a tree of arbitrary depth and fanout 4 executing the synchronous algorithm, if the condition

$$(5.10) \quad p > 66 + \sqrt{366^2 + 3p}$$

is met then no slave will ever be ready to send points to its master before that master is finished sending points to the other slaves.

M-Dimensional Problem Space

The tree architecture can easily be adapted to locally-defined iterative methods in M dimensions. We assume that the fanout of the processor tree is some perfect M th power, n^M . The height of the processor tree is q . We assume that the original problem grid is pn^q points on a side, with $(pn^q)^M$ points in all. Each master divides its region into n^M regions. Thus each terminal processor is assigned a region with p^M points. Using methods similar to those in the two-dimensional case, we can calculate a_q , the finishing time for one step, as

$$a_q = (n^M + 1)(qp + 246p^{M-1} \frac{n(M-1)q_{-1}}{n^{M-1}-1}) + p^M.$$

As in the two-dimensional case, we can prove that for a given number of terminal processors, 2^M is the optimal fanout among perfect M th powers.

Semi-Synchronous Method

We have seen that for a given number of leaf processors, the tree architecture is less efficient than the grid architecture. The cause of this inefficiency is that the leaf processors sit idle while masters higher in the tree exchange border points. This section investigates a technique for decreasing this inefficiency. The technique, called the Semi-Synchronous algorithm, is based on the observation that a slave need not perform all of the computation for one time step before sending border points to its master. After receiving border points, the slave can compute its border points first, then immediately send them to its master. Thus the communication of border points up and then down the processor tree has a head start and proceeds in parallel with the remaining computation on leaf processors. We want to give conditions under which the batch of border points for the next time step are ready for the slave when that slave is done with the current time step. From the slave's point of view, a time step starts when border points start arriving from its master. At time

$t = (p+46p)$, the slave finishes receiving its neighbors' border points and starts computing its own border points. After computing its border points, the slave requests to send them, and goes back to computing the rest of the points. When the master is ready to receive, the slave interrupts its computation and takes $p+46p$ units of time to send its $4p$ border points. After these points are sent, the slave resumes computing interior points. Hence the slave can complete its cycle for the semi-synchronous algorithm in

$$(5.11) \quad 2(p+46p) + p^2$$

units of time.

In the semi-synchronous method, we require that leaf slaves rush to send freshly computed border points to their master. The masters, by contrast, will execute the same algorithm as in the synchronous algorithm. We therefore assume that the fanout of the processor tree is four, since we have already seen that this fanout minimizes total exchange time. From the point of view of the master of a terminal slave, a time step starts when it has finished receiving border points from its master and attempts to send border points to its first slave. Since that slave may still be busy with computation from the previous time step, the master must wait a certain amount of time x . At time x , the master starts sending points to the first slave

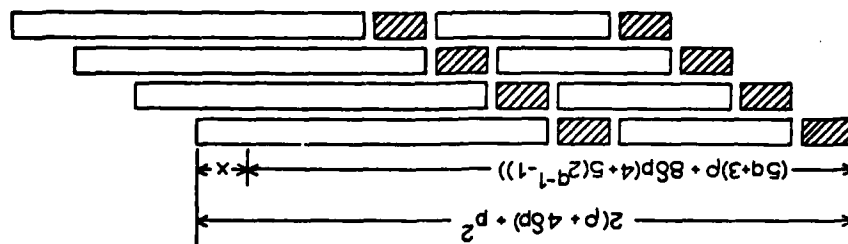


Fig. 5.5. One time step in the semi-synchronous method.

(Figure 5.5). At time $x + 4(p+46p)$, the master starts receiving points from the first slave, and at time $x + 8(p+46p)$ finishes receiving points from all the slaves. We assume that the first slave is ready to start sending points when the master is ready to start receiving them. This assumption is true when the computation time for the border points, $4p$, is less than or equal to the time to pass point values to the other three slaves, $3(p + 46p)$. The necessary condition is

$$3p + 4p(36-1) > 0.$$

Let b_k , for $k \geq 1$, be the time for a master at height k from the leaves to complete one time step. We have just shown that $b_1 = 8(p+46p)$. Masters above level one see the same behavior in their neighbors as in the synchronous algorithm. Hence

$$b_{k+1} = b_k + 5(p+46p2^k).$$

The solution to this recurrence relation for $k=q$ is

$$(5.12) \quad b_q = x + (5q+3)p + 86p(4+5(2^{q-1}-1)),$$

which is the finishing time of a processor tree of height q and fanout four that is executing the semi-synchronous algorithm, when masters of terminal slaves must wait time x to send points to their slaves.

When $x > 0$, the cycle time of the terminal slaves is greater than the cycle time of the rest of the processor tree, and so b_q also equals (5.11), the cycle time for a

slave:

$$(5.13) \quad x + (5q+3)p + 86p(4+5(2^q-1)) = 2(p+46p) + p^2.$$

With equation (5.13), the condition $x > 0$ becomes

$$p^2 - 86p(3+5(2^q-1)) - p(5(q-1)+6) > 0.$$

Hence when

$$(5.14) \quad p >$$

$$46(3+5(2^q-1)) + \sqrt{16(6(3+5(2^q-1)))^2 + p(5(q-1)+6)},$$

the slave processors are never idle, and the slave cycle time (5.11) represents the finishing time of the algorithm. When $x > 0$ we say that the tree is compute-bound. When (5.14) is not satisfied, (5.12) with $x=0$ gives the finishing time, and the tree is exchange-bound.

When the semi-synchronous method is compute bound, its finishing time, $2p + 86p + p^2$, is less than the finishing time of the grid topology, $8p + 86p + p^2$. In the semi-synchronous method a slave can accomplish all of its sending and receiving in 2 messages, but the grid-topology algorithm requires 8 messages, leading to the 6p time difference. Since the semi-synchronous method gives the exchange of border points a head start, it always finishes sooner than the synchronous tree method. The grid algorithm, the synchronous tree algorithm and the semi-synchronous method all give nearly n-fold speedup on large problems: The speedup in all three algorithms approaches the number of slave processors as the problem size goes to infinity.

5.4.3. Efficiency

Theorem 5.1 shows that the efficiency e of the grid algorithm is

$$e = \frac{p^2}{8p + 86p + p^2}.$$

Theorem 5.2 shows that the efficiency e of the synchronous tree algorithm is

$$e = \frac{p^2}{5(qp + 46p(2^q-1)) + p^2}.$$

The efficiency e of the semi-synchronous tree algorithm is

$$e = \frac{p^2}{2p + 86p + p^2}$$

if it is compute-bound. Solving each these three equations for p , and representing $e/(1-e)$ by W , we get

$$(5.15) \quad p = 46W + 2\sqrt{46^2W^2 + 2pW}$$

for the grid architecture,

$$(5.16) \quad p = 10W6(2^q-1) + \sqrt{(10W6(2^q-1))^2 + 5Wqp}$$

for the synchronous tree architecture, and

$$(5.17) \quad p = 46W + \sqrt{166^2W^2 + 2pW}$$

for the compute-bound semi-synchronous tree architecture.

For given q , p and q , these equations describe the minimum value of p for which the various algorithms will yield a given efficiency e . As the desired efficiency increases to one we must put a larger and larger subproblem on each leaf or grid processor.

5.4.4. Measurement of Communication Time

We have defined δ to be the ratio of per-point communication time to per-point computation time. In this section we present measurements taken on a VAX-11/780 and a PDP-11/70 that can be used to estimate δ for those machines. The following C subroutine was compiled on each

```
machine:
#define SIDE 50
double buffer[SIDE][SIDE];
GS(k) int k; {
    register double *p;
    register int j,i;
    double quart = 1/4.0;
    for(i=0; i<SIDE; i++) {
        buffer[0][i] = buffer[SIDE-1][i]
        = buffer[i][0] = buffer[i][SIDE-1] = 9.9;
    }
    for(i=k>0; k--){
        p = &buffer[1][1];
        for(i=1; i<SIDE-1; i++){
            for(j=1; j<SIDE-1; j++){
                *p = (*p - SIDE) + *(p + SIDE)
                    + *(p - 1) + *(p + 1)
                    + (quart);
                p++;
            }
            p += 2;
        }
    }
}
```

This subroutine implements the Gauss-Seidel method. We have gone to some effort to produce efficient code in this example. For example, we rejected array access by subscript because repeated address calculations produce a program about four times slower than the above. The C compiler on the VAX produced the following assembler code for the nested "i" and "j" loops at the end of the routine:

```
movl $1,r9          /initialize i loop
L5: movl $1,r10       /initialize j loop
L3: addd3 400(r11),-400(r11),r0 /south + north
    addd2 -8(r11),r0  /add west neighbor
    addd2 8(r11),r0   /add east neighbor
    mulf3 -12(fp),r0,(r11) /multiply by 1/4.0
    addl2 $8,r11       /move pointer 1 step east
    aobls $49,r10,L3   /j loop control
    addl2 $16,r11      /move pointer to next row
    aobls $49,r9,L5    /i loop control
```

The C compiler on the PDP-11/70 produced the following assembler code for the same C code segment:

```
mov $1,r3           /initialize i loop
L5: mov $1,r2        /initialize j loop
L3: movf -620(r4),r0 /add north neighbor
    addf -10(r4),r0  /add west neighbor
    addf 620(r4),r0  /add south neighbor
    addf 10(r4),r0   /add east neighbor
    mulf -20(t5),r0  /multiply by 1/4.0
    movf r0,(r4)     /store result
    add $10,r4       /move pointer 1 step east
    inc r2           /j loop control
    cmp $61,r2
    jgt L3
    add $20,r4       /move pointer to next row
    inc r3           /i loop control
    cmp $61,r3
    jgt L5
```

On both the VAX and PDP-11, the routine was executed with an argument of 500, so that the body of the "k" loop

was executed 500 times. Since "SIDB" was defined to be 50, the body of the "j" loop was executed $48 \times 48 \times 500$ times. Elapsed time on the VAX was 20.3 seconds, on the PDP-11 was 34.0 seconds. Time per point on the VAX is therefore $20.3 / (48 \times 48 \times 500) = 16.24$ microseconds, and on the PDP-11 is $34.0 / (48 \times 48 \times 500) = 27.2$ microseconds.

The DALL-B DMA Unibus link can transfer 500,000 16-bit words per second between two Unibuses. At this rate, the communication time for one 64-bit floating point word is 8 microseconds. If we use this hardware for connecting machines, $\delta_{VAX} = 0.49$ and $\delta_{PDP-11/70} = 0.29$.

We have defined ρ as the ratio of per-message overhead time to per-point computation time. Estimating ρ is harder than estimating δ because ρ depends on the operating system as well as the hardware. One distributed operating system for which we can estimate ρ is Arachne. Assuming 32-bit floating point numbers, computation time per point on an LSI-11 is approximately 500 microseconds. Per-message overhead on Arachne is approximately 12 milliseconds.

Hence $\rho_{Arachne} = 24$. Since time to send one floating-point word, ignoring per-message overhead, is about 0.4 milliseconds, $\delta_{Arachne} = 0.8$. These sample figures for ρ and δ are not necessarily representative of real distributed systems.

The values of ρ and δ strongly influence the efficiency of distributed systems in solving the Dirichlet problem. We can use these estimates to gain information about various topologies of LSI-11s running the Arachne distributed operating system. For example, these figures along with Theorem 5.5 require $p \geq 15$ in order to guarantee that sends and receives not overlap in the synchronous algorithm. For the semi-synchronous algorithm, relation (5.14) tells us that for trees of heights one and two, the relations $p \geq 54$ and $p \geq 118$ respectively must hold in order that the slave cycle time (5.11) would represent the finishing time. Otherwise the finishing time is represented by equation (5.12) with x set to zero. Relation (5.15) tells us that for the grid architecture to achieve an efficiency of 50%, $p \geq 18$ must hold. For the grid architecture to achieve an efficiency of 75%, p must be greater than or equal to 36. Relation (5.16) requires $p \geq 22$ and 53 for the synchronous algorithm running on trees of height one and two, respectively, to achieve an efficiency of 50%.

5.4.5. Scheduling Tree Machines

An entire tree machine need not be devoted to a single problem. For example, a tree of height 3 and fanout 4 can also be considered as 4 trees of height 2, 16 trees of height 1, or 64 trees of height 0 (serial processors).

Since we can never achieve n -fold speedup from n leaf processors, the efficiency of a tree machine is less than the efficiency of its workers, the leaf processors. Hence, if we have a large queue of problems, throughput is maximized by using the 4^q leaf processors as individual servers.

But throughput is not likely to be the primary concern of builders and users of parallel architectures. A parallel machine sacrifices low cost and efficiency to gain speed in the execution of lengthy tasks. As hardware becomes less expensive, this tradeoff becomes more profitable. Speed may be desired for various reasons. We may want to meet a time constraint (as in weather prediction or real-time applications), or we might simply want to minimize time spent by a human waiting for a computation. For such a machine to accomplish its purpose, the average inter-arrival time for tasks must be significantly greater than the average execution time. Otherwise a queue would form, defeating the original purpose, which is speed. Hence the scheduling of a tree machine is not likely to be a complicated issue. To achieve the primary goal of speed, a scheduling algorithm that devotes the entire tree to each task until completion would likely be satisfactory. During those periods when no lengthy tasks are requesting service, the tree-machine may be partitioned into individual processors for other tasks.

5.4.6. Static Non-Uniform Regions

As long as the problem grid is uniform and square, the grid architecture is sufficient and the tree architecture unnecessary. Unfortunately, many problems cannot be modeled in such a regular way. First, models of moving fluids often require extra grid points in places where values are fluctuating rapidly. Second, computation is often conditional. For example, a weather model may skip the calculation of some radiation terms at a point if clouds are present. Additional radiation terms may be omitted if it is night at a point. Third, the geometry of the problem space may be irregular. All of these situations create problems for a rectangular grid architecture, whether MIMD as discussed here, or SIMD as in the DAP [6].

The tree topology provides a solution to these problems by allowing load leveling through region encroachment. When load is evenly distributed throughout a square region, each master divides its region into 4 uniform smaller regions and assigns one region to each slave. When load is unevenly distributed, a master can divide its region into 4 equal-load smaller regions in the following way: First, divide the region along one dimension into two strips of equal load. Second, divide each of these strips along their long dimension into two regions of equal load. The 4 regions, called skewed quadrants, receive equal load (Fig-

ure 5.6). If each master follows this procedure, terminal slaves receive regions of equal load. Figures 5.7 through 5.10 illustrate this procedure carried out for a tree architecture of fanout four and height two, and for problem regions in the form of a disc, half-disc, annulus, and thin strip. Figure 5.11 illustrates the procedure carried out for a tree architecture of fanout four and height three when the load in the unit square is distributed according to $x^3 + y^3$.

5.4.7. Dynamic Region Encroachment

At the cost of $1/3$ more processors for communications, the tree topology gains the advantage of flexibility over the grid topology. As we have seen, this flexibility allows load-leveling by region encroachment among those processors actually doing the calculation. If the load function varies with time, this load-leveling can occur dynamically. For example, at each time step a master can decide if the load in its region is evenly divided among its four slaves by noting their computation times. Since a slave may spend part of its time rebalancing its own slaves, its master can discover the actual computation time only by being told by the slave. Thus each message bearing border points from a slave to its master should include the computation time for that slave. If a slave is overloaded, the

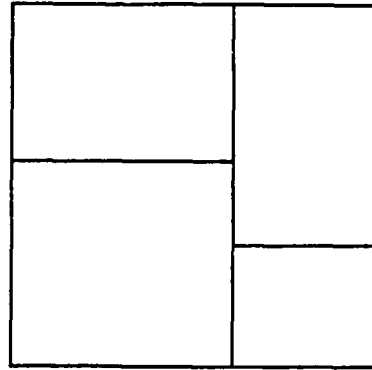


Fig. 5.6. Skewed quadrants.

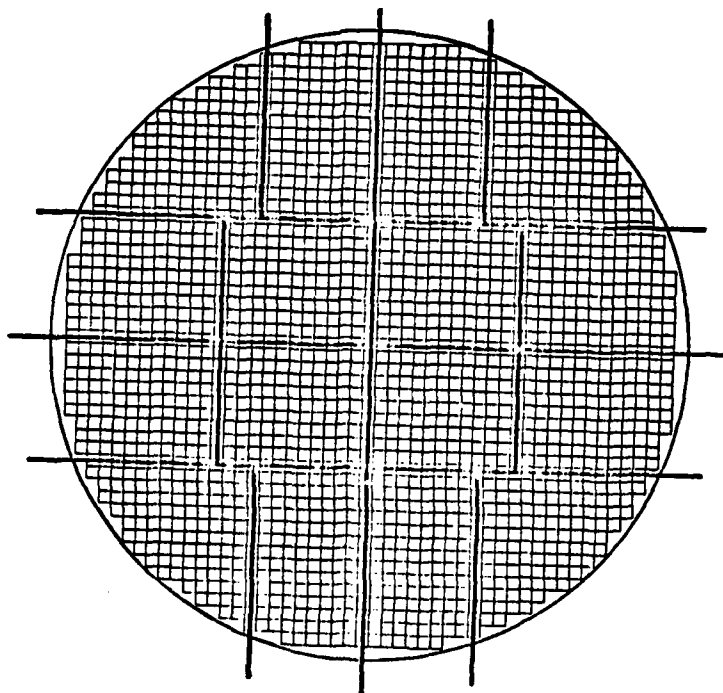


Fig. 5.7. Disc.

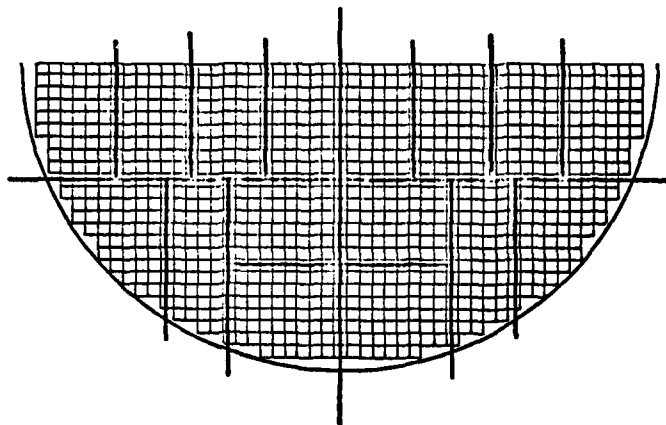


Fig. 5.8. Half-disc.

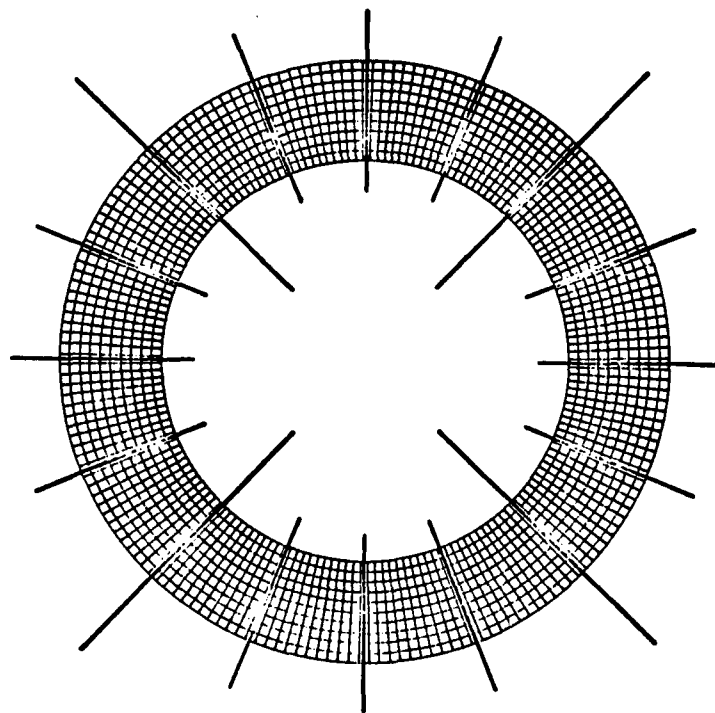


Fig. 5.9. Annulus.

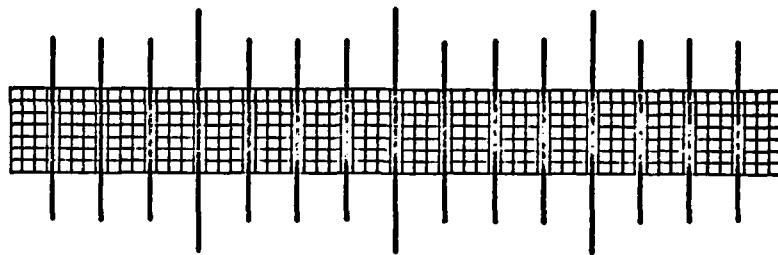


Fig. 5.10. Thin strip.

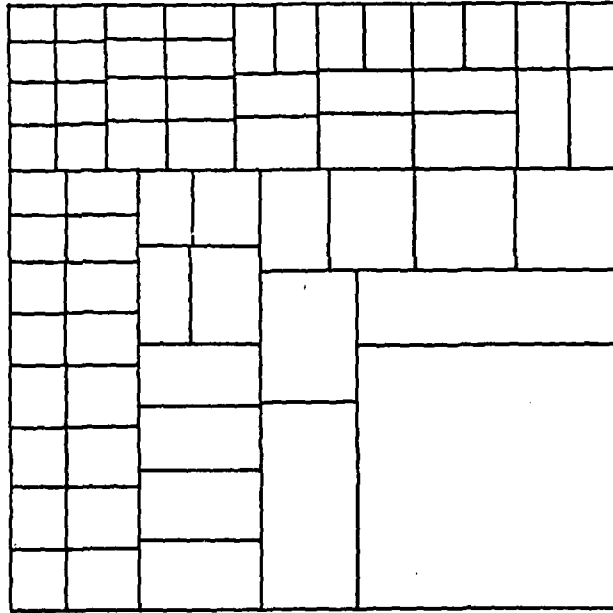


Fig. 5.11. Load = $x^3 + y^3$ on unit square.

master may decide to equalize load with a new set of skewed quadrants. The master then sends messages to those slaves that will lose points and receives in reply the points that will be transferred. These points are then forwarded to the appropriate slaves, after which the normal sequence of events resumes. Since the load-leveling itself entails a cost, a master must weigh that cost against the expected speedup. Load-leveling between slaves would occur only when imbalance exceeds a certain threshold.

Since a master can also be a slave, it must also be prepared for messages from its master that give or take away points. A master M follows the following algorithm:

1. Receive message from master.
2. If the message is a request for points, forward it to the affected slaves and assemble their replies into a reply to the master. If the message is a set of points to be added to M 's domain, divide the points into one package for each affected slave and relay the packages to these slaves. Go to 1).
3. If the message is the usual update of border points, relay these updates to slaves. Go to 1).
4. Decide whether load-leveling needs to be done among the slaves. If load-leveling is needed, send messages to slaves that lose points and relay the points in their replies to the appropriate siblings. Go to

1).

Even if the load does not vary with time, dynamic region encroachment could be used to adapt automatically to an arbitrary uneven load. The regions could initially be as for an even load, and would converge toward a configuration with equal loads.

Figure 5.12 illustrates dynamic region encroachment for a region in which the load is a bivariate normal distribution on the square $(-5,5) \times (-5,5)$ with a time-varying standard deviation.

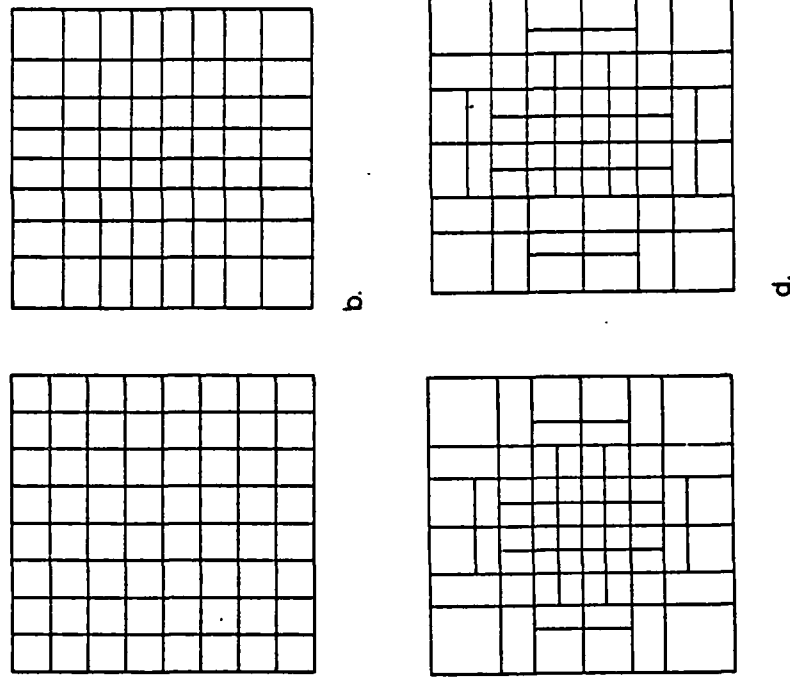


Fig. 5.12. Load = bivariate normal distribution with standard deviation a) 100; b) 4; c) 3; d) 2.5.

Chapter 6 - Quotient Networks

It is a profoundly erroneous truism, repeated by all copy-books and by eminent people when they are making speeches, that we should cultivate the habit of thinking what we are doing. The precise opposite is the case. Civilization advances by extending the number of important operations which we can perform without thinking about them.

Alfred North Whitehead
quoted in A Certain World
by W. H. Auden

6.1. INTRODUCTION

One barrier to the practical use of interconnection networks is the lack of algorithms for processing large problems on small machines. (By an interconnection network we mean an SIMD parallel computer interconnected by some interconnection strategy.) Often, it is assumed that N processors are available to process N data [13,49]. If we happen to have $N+1$ or more data points, then we must choose between the serial algorithm and a bigger machine. Examples of interesting exceptions can be found in work by Baudet and Stevenson [21], and by Siegel, Mueller and Siegel [50]. This chapter investigates a method that constructs algorithms for solving large problems on small net-

works. We call these algorithms quotient-network algorithms. In Section 2, we review some proposed interconnection networks. Section 3 reviews proposed algorithms for those networks. We call these algorithms large-network algorithms, since each one assumes as many processors as points in the problem to be solved. Section 4 presents a general method for transforming a large-network algorithm into a quotient-network algorithm. Section 5 applies this method to each of the algorithm-machine combinations of Section 3. Section 6 discusses some economic advantages of quotient-network algorithms.

6.2. EXISTING NETWORKS

In this section we briefly review some proposed interconnection networks. For a more thorough overview, see [51]. We assume that each network contains N processors. We denote the square root of N by n , and $\log_2 N$ by m . We will name the processors $PE(0)$ through $PE(N-1)$. Sometimes we refer to a processor by the binary form of its number, $p = p_{m-1}p_{m-2}\dots p_1p_0$.

6.2.1. Grid-Connected Network

In this network, the processors are arranged in a two-dimensional n by n grid. The processor in the i th row

and j^{th} column is named $PE(i, j)$, for $0 \leq i, j < n$. A processor is connected to its north, south, east and west neighbors:

If $i > 0$, $PE(i, j)$ is connected to $PE(i-1, j)$.
 If $i < n-1$, $PE(i, j)$ is connected to $PE(i+1, j)$.
 If $j > 0$, $PE(i, j)$ is connected to $PE(i, j-1)$.
 If $j < n-1$, $PE(i, j)$ is connected to $PE(i, j+1)$.

The Illiac IV network adds additional connections among edge processors [2].

6.2.2. Perfect Shuffle

Shuffle-Exchange

In this network, $PE(p_{m-1}p_{m-2} \dots p_1 p_0)$ is connected to $PE(p_{m-2} \dots p_1 p_0 p_{m-1})$ by the "shuffle function" line and to $PE(p_{m-1}p_{m-2} \dots p_1 \bar{p}_0)$ by the "exchange function" line.

4-Pin Shuffle

In this network, each processor has two input pins $IPIN0$ and $IPIN1$, and two output pins $OPIN0$ and $OPIN1$. We can number all input pins by assigning to $IPIN0$ on processor $p_{m-1}p_{m-2} \dots p_1 p_0$ the number $p_{m-1}p_{m-2} \dots p_1 p_0$. $IPIN1$ on the same processor is assigned the number $p_{m-1}p_{m-2} \dots p_1 p_0^1$. This numbering allows us to refer to input pins as $IPIN(0)$ through $IPIN(2N-1)$. Output pins are numbered in the same way, $OPIN(0)$ through $OPIN(2N-1)$. The shuffle function is used to transfer data from the output pins of $PE(p_{m-1}p_{m-2} \dots p_1 p_0)$ to the input pins of processors

$PE(p_{m-2} \dots p_1 p_0)$ and $PE(p_{m-2} \dots p_1 p_0^1)$. (The name "4-pin shuffle" is new.)

6.2.3. PM2I

In the Plus-Minus 2^i network (PM2I), $PE(j)$ is connected to processors

$$j+2^i \bmod N$$

and

$$j-2^i \bmod N,$$

for $0 \leq i < m$.

6.2.4. Cube

$PE(p_{m-1} \dots p_{i+1} p_i p_{i-1} \dots p_0)$ in the cube network is connected to the m processors $PE(p_{m-1} \dots p_{i+1} \bar{p}_i p_{i-1} \dots p_0)$, for $0 \leq i < m$.

6.3. EXISTING ALGORITHMS

In this section we review some proposed large-network algorithms. The number of such algorithms is large and growing, so we do not attempt to be comprehensive. Our goal is to illustrate the process of transforming large-network algorithms into quotient-network algorithms.

6.3.1. Fast-Fourier Transform on the Shuffle

Let $A(k)$, $k=0, 1, \dots, N-1$, be a vector of N complex numbers. The Discrete Fourier Transform (DFT) of A is defined to be the vector

$$(6.1) \quad X(j) = \sum_{k=0}^{N-1} A(k)W^{jk} \quad j=0,1,\dots,N-1$$

where $W = e^{2\pi i/N}$. The obvious algorithm for computing X takes time $O(N^2)$. An important advance in the theory of algorithms was the discovery of an $O(N \log N)$ algorithm for the DFT [24]. This algorithm is called the Fast Fourier Transform (FFT). Pease [10] has discovered an algorithm

that computes the DFT on $N/2$ processors in time proportional to $\log N$, thus achieving optimal speedup. Pease's algorithm can be explained as follows: First, we represent both k and j by their binary expansion.

$$k = k_{m-1}k_{m-2}\dots k_0$$

$$j = j_{m-1}j_{m-2}\dots j_0$$

Equation 6.1 then becomes

$$(6.2) \quad X(j) = \sum_{k_0} \sum_{k_1} \dots \sum_{k_{m-1}} A(k_{m-1}k_{m-2}\dots k_0)W^{jk_{m-1}2^{m-1} + jk_{m-2}2^{m-2} + \dots + jk_0}$$

$$= \sum_{k_0} W^{jk_0} \sum_{k_1} W^{jk_12} \dots \sum_{k_{m-1}} W^{jk_{m-1}2^{m-1}} A(k_{m-1}k_{m-2}\dots k_0).$$

Equation 6.2 consists of m nested summations. Since W^{N-1} ,

$$W^{j2^{m-s}} = W^{js-1js-2\dots j_0 \cdot 2^{m-s}},$$

so the innermost s summations depend only on the m binary variables $j_0, \dots, j_{s-1}$ and $k_{m-s-1}, \dots, k_0$. Thus the innermost s summations represent a function from $0, \dots, N-1$ to the complex numbers; we represent this function as an array B_s of N complex numbers. B_s satisfies

$$B_0(k_{m-1}k_{m-2}\dots k_0) = A(k_{m-1}k_{m-2}\dots k_0),$$

$$(6.3) \quad B_s(j_0\dots j_{s-1}k_{m-s-1}\dots k_0) =$$

$$\sum_{k_{m-s}} B_{s-1}(j_0\dots j_{s-2}k_{m-s}\dots k_0)W^{j_{s-1}j_{s-2}\dots j_0 \cdot 2^{m-s} \cdot k_{m-s}},$$

and

$$B_m(j_0\dots j_{m-1}) = X(j_{m-1}\dots j_0).$$

Equation 6.3 reveals how we can use the 4-pin shuffle to compute the DFT: We iteratively compute B_s for $s = 1$ to m . Iteration s results in B_s distributed on the output pins. To perform iteration s , we form the weighted sum of elements from B_{s-1} whose indices differ only in bit position number $m-s$. The 4-pin shuffle with $N/2$ processors provides exactly the data alignment we want, since shuffling an array s times causes the indices of the two numbers in each processor to differ only in bit position number $m-s$.

The following is a description of Pease's parallel FFT algorithm. The hardware is assumed to be a 4-pin shuffle with $N/2$ PEs. The machine operates in SIMD mode, and PEs differ only in that each processor $PE(p_{m-2}\dots p_0)$ knows its own address $p_{m-2}\dots p_0$.

Large-network Parallel FFT

Input: data items $A(k)$ $k=0, \dots, N-1$
with $A(k)$ on $OPIN(k)$

Output: the Fourier transform $X(j)$ of $A(k)$
with $X(j_{m-1} \dots j_0)$ on $OPIN(j_0 \dots j_{m-1})$

```

for s := 1 to m
begin
  SHUFFLE,
     $OP_0 \dots P_{s-2} \cdot 2^{m-s}$ 
     $OPIN_0 := IPIN_0 + W_{IP_0 \dots P_{s-2} \cdot 2^{m-s}} \cdot IPIN_1$ 
     $OPIN_1 := IPIN_0 + W_{IP_0 \dots P_{s-2} \cdot 2^{m-s}} \cdot IPIN_1$ 
end

```

This algorithm can be proved correct by induction on the following loop invariant:

Immediately after shuffle number s ,

and $B_{s-1}(j_0 \dots j_{s-2} 0^k_{m-s-1} \dots k_0)$
and $B_{s-1}(j_0 \dots j_{s-2} 1^k_{m-s-1} \dots k_0)$

are in processor $PE(k_{m-s-1} \dots k_0 j_0 \dots j_{s-2})$ at pin positions

and $IPIN(k_{m-s-1} \dots k_0 j_0 \dots j_{s-2} 0)$
and $IPIN(k_{m-s-1} \dots k_0 j_0 \dots j_{s-2} 1)$,

respectively. This processor then places

and $B_s(j_0 \dots j_{s-2} 0^k_{m-s-1} \dots k_0)$
and $B_s(j_0 \dots j_{s-2} 1^k_{m-s-1} \dots k_0)$

onto output pin positions

and $OPIN(k_{m-s-1} \dots k_0 j_0 \dots j_{s-2} 0)$
and $OPIN(k_{m-s-1} \dots k_0 j_0 \dots j_{s-2} 1)$,
respectively.

6.3.2. Sorting on the Shuffle

Batcher's algorithm [11], as adapted by Stone [13], sorts N numbers in $\log^2 N$ passes through the $N/2$ -processor 4-pin shuffle. After each shuffle, a processor either

1. Copies the two inputs directly to the two outputs.
2. Compares the two inputs and puts the lower on $OPIN_0$ and the higher on $OPIN_1$.
3. Compares the two inputs and puts the higher on $OPIN_0$ and the lower on $OPIN_1$.

Hence Batcher's algorithm requires $\log^2 N$ shuffle steps on the 4-pin shuffle.

6.3.3. Polynomial Evaluation on the Shuffle

Consider the problem of evaluating the $(N-2)$ nd-degree polynomial

$$(6.4) \quad \sum_{i=0}^{N-2} a_i x^i$$

for given numbers x and a_i , $i=0, \dots, N-2$. Horner's rule, which evaluates a polynomial by the scheme

$$(\dots((a_n x + a_{n-1})x + \dots + a_1)x + a_0,$$

is an optimal serial algorithm that requires exactly $N-2$ multiplications and $N-2$ additions. Stone [13] presents an algorithm for computing (6.4) with $2 \log N$ passes through the $N/2$ -processor 4-pin shuffle.

6.3.4. Finite-difference Methods

The literature is full of proposals for the parallel execution of finite-difference calculations [6,28,44,46,47,48]. Often, the rectilinear problem grid is mapped one-to-one onto the rectilinear processor grid. At each time step, each processor communicates its values to and receives values from each of its nearest neighbors. This exchange provides each processor with the necessary values to compute the value of its point at the next time step.

6.4. NETWORK EMULATION

Definition. Suppose that $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are graphs. We say that a function $f: V_H \rightarrow V_G$ is an emulation of H by G if for every edge $(h_1, h_2) \in E_H$

$$f(h_1) = f(h_2) \text{ or } (f(h_1), f(h_2)) \in E_G.$$

Every emulation $f: V_H \rightarrow V_G$ induces a mapping $f': E_H \rightarrow V_G \cup E_G$ in a natural way:

$$f'(h_1, h_2) = (f(h_1), f(h_2)) \text{ if } (f(h_1), f(h_2)) \in E_G \\ \text{otherwise}$$

$$f'(h_1, h_2) = f(h_1) = f(h_2).$$

We say that the node $g \in V_G$ emulates the nodes $f^{-1}(g)$, and that the edge $(g_1, g_2) \in E_G$ emulates the edges $f^{-1}(g_1, g_2)$. If $|f^{-1}(g)|$ is the same for every $g \in V_G$, then we say that f is computationally uniform, and $|f^{-1}(g)|$ is the computa-

tion factor of f , if $|f^{-1}(e)|$ is the same for every $e \in E_G$, then we say that f is exchange-uniform, and $|f^{-1}(e)|$ is the exchange factor of f . If f is computationally uniform and exchange-uniform, and if the computation factor equals the exchange factor, then we say that f is totally uniform, and $|f^{-1}(g)|$ is the emulation factor of f .

If the graphs G and H are interconnection networks, then the existence of an emulation of H by G provides a way for the network G to emulate the actions of the network H . By analogy with the notion of quotient groups in abstract algebra, we call G a quotient network. The processor $g \in V_G$ is time-shared to emulate the group of processors $f^{-1}(g)$ in V_H , and the communications line $(g_1, g_2) \in E_G$ is time-multiplexed to emulate the communication lines $f^{-1}(g_1, g_2)$ in E_H .

If the emulation of H by G is computationally uniform, then the processors in G can efficiently perform the actions of the processors of H : Since each processor in G emulates the same number of processors of H , all of the processors in G can proceed in unison and finish simultaneously. No processors sit idle while other overloaded processors finish their work. Likewise, if the emulation of H by G is exchange-uniform, then the communications lines in G can efficiently perform the actions of the communications lines of H : Since each communications line in G emulates

the same number of communications lines of H , all of the data transfers in G can proceed in unison and finish simultaneously. No communications lines sit idle while other overloaded communications lines finish their work.

We now present an emulation for each of the networks reviewed in Section 2. In each case, a large network H is emulated by a smaller network G of the same general interconnection scheme.

6.4.1. Perfect Shuffle

Suppose that H is a shuffle-exchange network with $N = 2^m$ processors. We will emulate this network with a 4-pin shuffle network of size $N/2$, and then emulate the 4-pin shuffle network with any 4-pin shuffle network of size a smaller power of two.

Theorem 6.1. The function $f(p_{m-1} \dots p_2 p_1 p_0) = p_{m-1} \dots p_2 p_1$ emulates the shuffle-exchange network of size N with the 4-pin shuffle of size $N/2$.

Proof. Suppose that $e = (h_1, h_2) \in E_H$. If e is an exchange connection, then $f(h_1) = f(h_2)$. If e is a shuffle connection, then

$$\begin{aligned} & (f(h_1), f(h_2)) \\ &= (f(p_{m-1} \dots p_1 p_0), f(p_{m-2} \dots p_1 p_0 p_{m-1})) \\ &= (p_{m-1} \dots p_1, p_{m-2} \dots p_1 p_0) \in E_G. \end{aligned}$$

Q.E.D.

The emulation f is computationally uniform and exchange-uniform, but not totally uniform. The computation factor is two and the exchange factor is one.

Theorem 6.2. The function

$$\begin{aligned} f(p_{m+q-1} p_{m+q-2} \dots p_q p_{q-1} \dots p_0) &= p_{m+q-1} p_{m+q-2} \dots p_q \\ &\text{emulates the 4-pin shuffle of size } N = 2^{m+q} \text{ with the 4-pin} \\ &\text{shuffle of size } N = 2^m. \end{aligned}$$

Proof. Let

$$\begin{aligned} & (h_1, h_2) \\ &= (p_{m+q-1} p_{m+q-2} \dots p_q p_{q-1} \dots p_0, p_{m+q-2} \dots p_q p_{q-1} \dots p_0 x) \\ &\text{be an edge in } H. \text{ Then} \\ & (f(h_1), f(h_2)) \\ &= (p_{m+q-1} p_{m+q-2} \dots p_q, p_{m+q-2} \dots p_q p_{q-1}) \in E_G. \end{aligned}$$

Q.E.D.

The emulation f is totally uniform, with emulation factor 2^q. Figure 6.1 illustrates a 4-pin shuffle with four PEs emulating a 4-pin shuffle with eight PEs.

6.4.2. Grid-connected Network

The emulation of a large grid-connected network with a small one is fairly straightforward; we simply partition the large network into square regions.

Theorem 6.3. The function

$$\begin{aligned} f(p_{r+s-1} \dots p_r p_{r-1} \dots p_0, q_{r+s-1} \dots q_r q_{r-1} \dots q_0) \\ &= (p_{r+s-1} \dots p_r, q_{r+s-1} \dots q_r) \end{aligned}$$

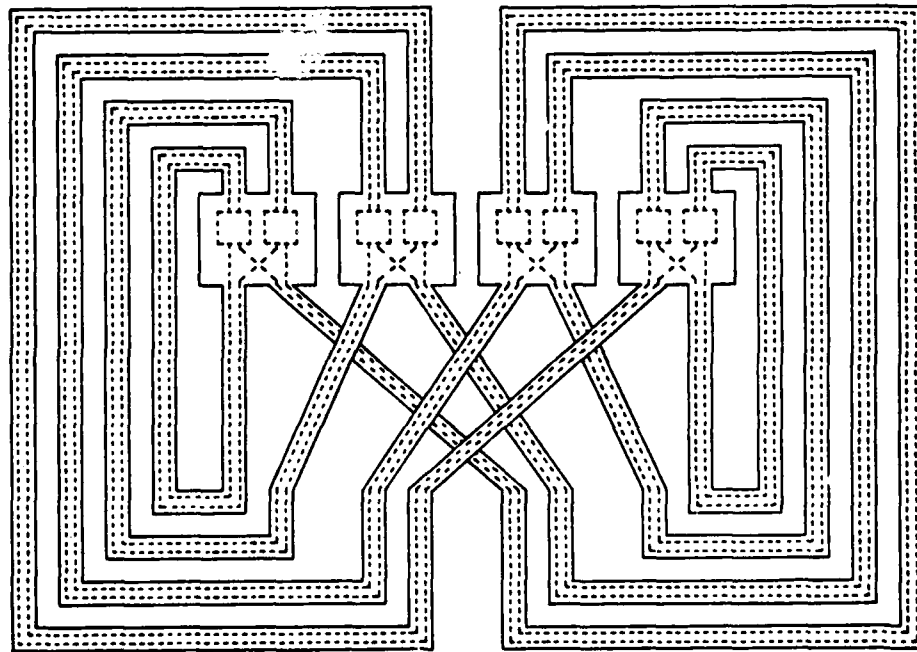


Fig. 6.1. 4-pin shuffle emulating 8-pin shuffle.

is an emulation of a grid-connected network of size 2^{2t+2s} by a grid-connected network of size 2^{2s} .

Proof. Suppose that $h = (h_1, h_2) = ((P_1, Q_1), (P_2, Q_2)) \in E_H$. We assume that h is a "North" connection, so that $P_1 = P_2$ and $Q_1 = Q_2 - 1$. A similar proof can be used when h is any of the other three grid connections. We can represent P_1 , P_2 , Q_1 , and Q_2 as follows:

$$P_1 = P_2 = P_{r+s-1} \dots P_r P_{r-1} \dots P_0$$

$$Q_1 = Q_2 - 1 = q_{r+s-1} \dots q_r q_{r-1} \dots q_0$$

If incrementing Q_1 results in a carry into the top s bits in its binary representation, then

$$(f(h_1), f(h_2))$$

$$= (f(P_{r+s-1} \dots P_r P_{r-1} \dots P_0, q_{r+s-1} \dots q_r q_{r-1} \dots q_0),$$

$$f(P_{r+s-1} \dots P_r P_{r-1} \dots P_0, q_{r+s-1} \dots q_r q_{r-1} \dots q_0 + 1))$$

$$= ((P_{r+s-1} \dots P_r, q_{r+s-1} \dots q_r),$$

$$(P_{r+s-1} \dots P_r, q_{r+s-1} \dots q_r + 1))$$

$$\in E_G.$$

If not, then

$$f(h_1)$$

$$= f(P_1, Q_1)$$

$$= f(P_{r+s-1} \dots P_r P_{r-1} \dots P_0, q_{r+s-1} \dots q_r q_{r-1} \dots q_0)$$

$$= (P_{r+s-1} \dots P_r, q_{r+s-1} \dots q_r)$$

$$= f(P_{r+s-1} \dots P_r P_{r-1} \dots P_0, q_{r+s-1} \dots q_r q_{r-1} \dots q_0 + 1)$$

$$= f(P_2, Q_2)$$

$$= f(h_2).$$

Q.E.D.

The emulation f is computationally uniform and exchange-uniform, but not totally uniform. The computation factor is 2^{2r} and the exchange factor is 2^r . Figure 6.2 illustrates part of a grid-connected network emulating a grid-connected network that is four times larger.

In general, a k -dimensional grid may be emulated by a smaller k -dimensional grid. For a given r , the exchange factor is 2^r and the computation factor is 2^{kr} .

6.4.3. Cube

Theorem 6.4. The function

$f(p_{m+q-1}p_{m+q-2}\dots p_q p_{q-1}\dots p_0) = p_{m+q-1}p_{m+q-2}\dots p_q$ emulates the cube of size $N=2^{m+q}$ with the cube of size $N=2^m$.

Proof. Suppose that $(h_1, h_2) \in E_N$. Then h_1 and h_2 are of the form

$$h_1 = p_{m+q-1}\dots p_1\dots p_0$$

$$\text{and } h_2 = p_{m+q-1}\dots \bar{p}_1\dots p_0.$$

If $i \leq q$, then $f(h_1) = f(h_2)$. If $i > q$, then

$$f(h_1) = p_{m+q-1}\dots p_1\dots p_q$$

$$\text{and } f(h_2) = p_{m+q-1}\dots \bar{p}_1\dots p_q.$$

Hence, $(f(h_1), f(h_2)) \in E_G$.

Q.E.D.

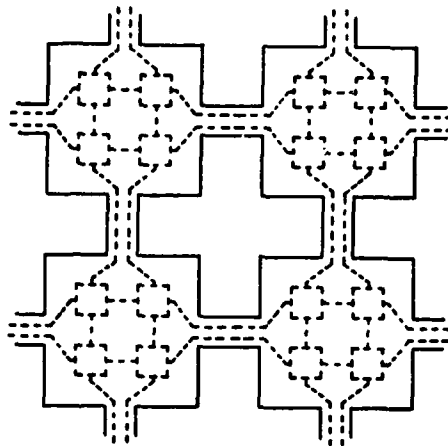


Fig. 6.2. Grid emulation with $r=2$.

AD-A123 586

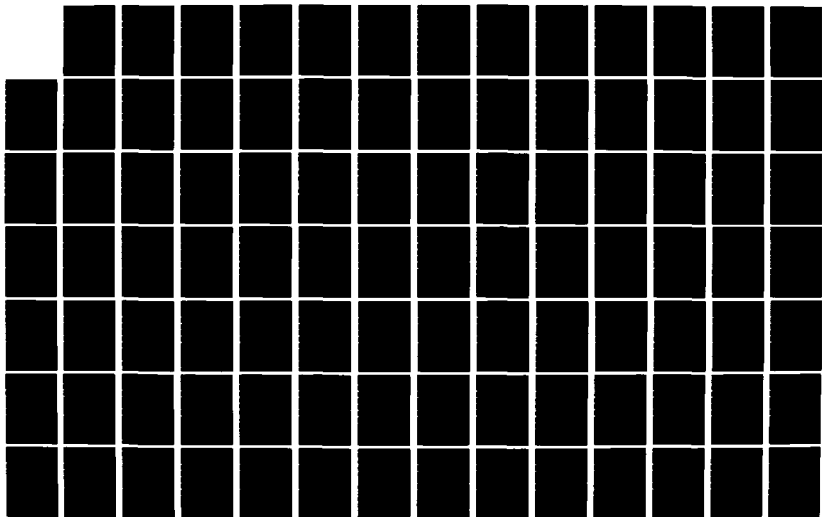
OPERATING SYSTEMS FOR RING-BASED MULTIPROCESSORS(U)
WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES
R A FINKEL ET AL. 1982 N00014-81-C-2151

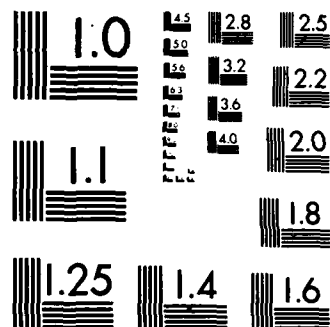
2/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

The emulation f is totally uniform, with emulation factor 2^q . Any function that discards q bits and permutes the remaining bits is also an emulation. Figure 6.3 illustrates a cube of size four emulating a cube of size eight.

6.4.4. PM2I

Theorem 6.5. The function

$f(p_{m+q-1}p_{m+q-2}\dots p_{q-1}\dots p_0) = p_{m+q-1}p_{m+q-2}\dots p_q$ emulates the PM2I network of size $NP=2^{m+q}$ with the PM2I network of size $N=2^m$.

Proof. Let $(h_1, h_2) \in E_H$. Hence h_1 and h_2 are of the form

$$h_1 = p_{m+q-1}p_{m+q-2}\dots p_q p_{q-1}\dots p_0$$

$$h_2 = p_{m+q-1}p_{m+q-2}\dots p_q p_{q-1}\dots p_0 + 2^i,$$

for some $0 \leq i < m$. If $i < q$ and if the addition of 2^i to h_1 does not cause a carry into the top m bits of its address, then $f(h_1) = f(h_2)$. Otherwise, if $i \geq q$ then

$$f(h_2) = f(h_1) + 2^{i-q},$$

and if $i < q$ then

$$f(h_2) = f(h_1) + 1.$$

In either case, $(f(h_1), f(h_2)) \in E_G$.

Q.E.D.

The emulation f is computationally uniform, with computation factor 2^q . But f is not exchange-uniform, since each "+1" link in G emulates $2^{m+1}-1$ links in H , while every other link in G emulates 2^m links. Figure 6.4 illustrates a

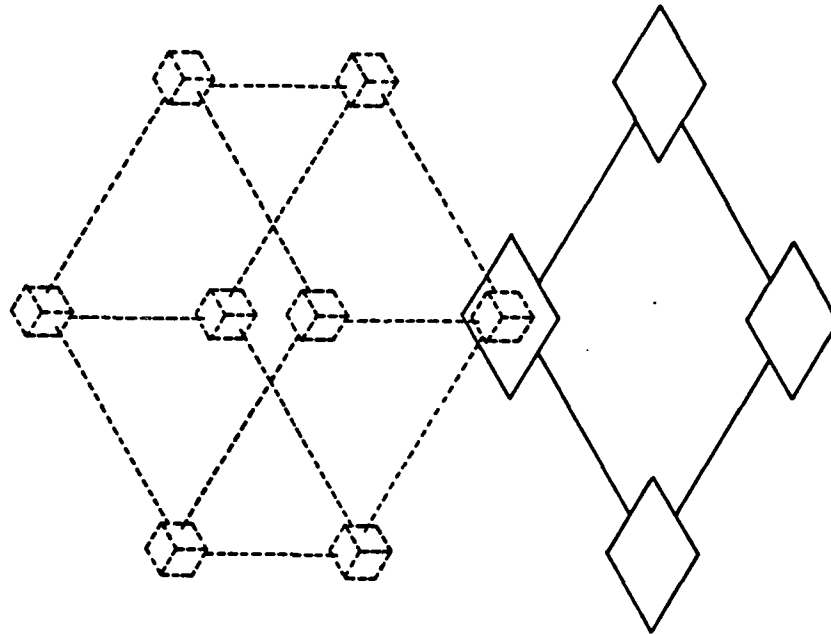


Fig. 6.3. Two-dimensional cube emulating three-dimensional cube.

PM2I of size eight emulating a PM2I of size sixteen.

6.5. SOME RESULTING ALGORITHMS

In this section we transform the large-network algorithms of Section 3 into quotient-network algorithms. Since the transformation is a fairly mechanical one, we present it in detail only for the FFT algorithm. For the other algorithms we only summarize the result.

6.5.1. Fast-Fourier Transform on the Shuffle

The FFT algorithm presented in Section 3 consists of a loop executed m times. The body of the loop consists of a SHUFFLE followed by placing weighted sums of the input pins onto the output pins. We assume, as in Section 3, that our network contains $N/2 = 2^{m-1}$ machines. We wish to compute the DFT of $N \cdot p = 2^{m+q}$ data items $A(i)$ for $i=0, \dots, N \cdot p-1$. We therefore emulate the actions of a 4-pin network of size $N \cdot p/2 = 2^{m+q-1}$. Each processor represents the virtual pins of the processors it is emulating by arrays: The virtual $PE(k_{m+q-2} \dots k_0)$ has pins $OPIN_0, OPIN_1, IPIN_0$, and $IPIN_1$. These pins are emulated on actual $PE(k_{m+q-2} \dots k_q)$ at index $k_{q-1} \dots k_0$ of arrays $EOPIN_0, EOPIN_1, EIPIN_0$, and $EIPIN_1$, respectively. Here is the quotient-network FFT algorithm:

Quotient-Network Parallel FFT

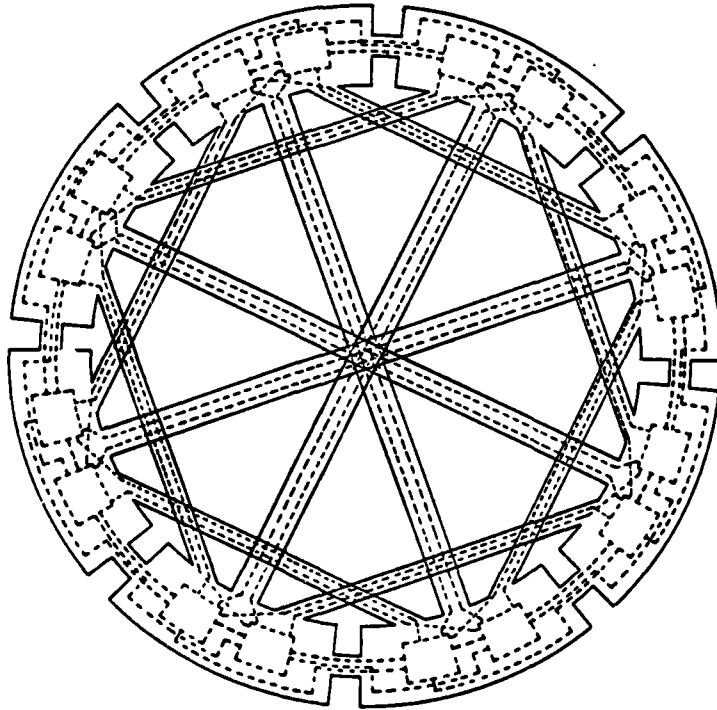


Fig. 6.4. PM2I of size eight emulating PM2I of size sixteen.

Input: data items $A(i)$, $i=0, \dots, M \cdot P-1$ such that

$A(k_{m+q-1} \dots k_{q+1} k_q \dots k_0)$ is stored on machine
 $PE(k_{m+q-1} \dots k_{q+1})$ in $EOPINK_0(k_q \dots k_1)$

Output: The Discrete Fourier Transform $X(i)$,
 $i=0, \dots, M \cdot P-1$ such that

$X(k_0 \dots k_{q+1} \dots k_{m+q-1})$ is stored on machine
 $PE(k_{m+q-1} \dots k_{q+1})$ in $EOPINK_0(k_q \dots k_1)$

for $s := 1$ to m do
begin

 { emulate SHUFFLE: }

 for $j := 0$ to $2^q - 1$ do
 begin { emulate $PE(P_{m-2} \dots P_0 j_{q-1} \dots j_0)$ }

 if j is even then

 begin

$OPINO := EOPINO(j/2); q-1;$

 end else

$OPINO := EOPINO(j/2+2^{q-1});$

 begin

$OPINO := EOPINO(\lfloor j/2 \rfloor); q-1;$

$OPINI := EOPINI(\lfloor j/2 \rfloor + 2^{q-1});$

 end;

 SHUFFLE;

$EIPINO[j] := IPINO;$

$EIPINI[j] := IPINI;$

 end;

 { emulate computation: }

 for $j := 0$ to $2^q - 1$ do

 begin { emulate $PE(P_{m-2} \dots P_0 j_{q-1} \dots j_0)$ }

$e_{m+q-2} \dots e_0 := P_{m-2} \dots P_0 j_{q-1} \dots j_0;$

$EOPINO[j] :=$

$e_0 \dots e_{s-2} \cdot 2^{m-s}$

$\cdot EIPINI[j];$

$EOPINI[j] :=$

$1e_0 \dots e_{s-2} \cdot 2^{m-s}$

$\cdot EIPINI[j];$

 end;

end;

end;

The original large-network FFT algorithm is optimal in the number of processors; that is, the speedup is proportional to the number of processors used. In the above example, the large-network FFT algorithm rests on top of an emulation, which rests on the actual hardware. Apart from the loop and indexing overhead needed to emulate the SHUFFLE step, the 2^{m-1} -processor network is 2^q times as slow as the 2^{m+q-1} -processor network. The loop and indexing overhead slows the algorithm down by only a constant factor, and could be eliminated entirely by unrolling the loops. Therefore the quotient-network algorithm is also optimal: We gain approximately N speedup with N processors. While the original large-network FFT algorithm performs $\log N$ operate-shuffle steps, the quotient-network algorithm performs $2^q \log N$ operate-shuffle steps.

6.5.2. Sorting on the Shuffle

As we mentioned above, Batchier's large-network algorithm sorts $N = 2^{m+q}$ numbers in $(m+q)^2$ operate-shuffle steps on a 4-pin shuffle with 2^{m+q-1} processors. A quotient-network version of this algorithm sorts the 2^{m+q} numbers in $2^q(m+q)^2$ operate-shuffle steps on a 4-pin shuffle with 2^{m-1} processors.

6.5.3. Polynomial Evaluation on the Shuffle

A 2^{m+q-1} -processor 4-pin shuffle network can evaluate a polynomial of degree $2^{m+q-1}-2$ in $2(m+q)$ operate-shuffle steps. The quotient-network version of this algorithm evaluates the same polynomial in $2^{q+1}(m+q)$ operate-shuffle steps with 2^{m-1} processors.

6.5.4. Finite-difference Methods

A large network algorithm that maps the 2^{2r+2s} points of a finite-difference grid one-to-one onto a 2^{2r+2s} grid-connected network must communicate each point at each time step to all four neighbors. The quotient-network version of this algorithm reduces the communication/computation ratio by communicating only the border points of a processor's region to that processor's neighbors. The grid algorithm of Chapter 5 can now be seen as a quotient-network algorithm.

6.5.5. Alpha-beta Search

One optimization of the Tree-splitting Algorithm mapped the top several layers of masters onto a single processor. For example, the root master and its two slaves can be processes on the root node of a processor tree. We can view this processor tree as a quotient network: The root node emulates the top three processors of a binary

processor tree.

6.6. THE ECONOMICS OF EMULATION

We can give several economic arguments in favor of solving large problems on small networks through emulation.

1. The cost of a word of storage is much smaller than the cost of a processor. This fact is independent of further increases in scale of integration. By adding extra storage at each processor in G, we increase the potential computation factor of an emulation; that is, we can emulate a larger H. We therefore increase the largest problem that the network can handle, with a much smaller increase in hardware cost than would be incurred by expanding G to H.
2. Suppose that a solution must meet a time constraint for a problem of size N. One processor cannot meet this constraint, but N processors (the network H) are much too expensive and much faster than needed. An intermediate number of processors (the network G) emulating H may be fast enough and affordable.
3. Given a large-network algorithm, an emulation automatically produces a quotient-network algorithm to solve the same problem on a smaller machine. We achieve economy of thought by solving once and for

all the "emulation problem": doing on a small machine what a large machine can do. Thereafter, we can deal exclusively with the simpler class of problems: data sets of size N on networks with N processors.

Chapter 7 - Conclusions and Future Directions

In literature, in art, in life, I think that the only conclusions worth coming to are one's own conclusions. If they march with the verdict of the connoisseurs, so much the better for connoisseurs; if they do not so march, so much the better for oneself.

- A.C. Benson
From a College Window

In this chapter we will briefly review the major contributions of this thesis. Where appropriate, we will indicate areas that need further research.

7.1. ALPHA-BETA SEARCH

We have presented two parallel algorithms for implementing alpha-beta search on a tree of processors. The first, Palphabeta, divides work recursively among slave processors in a simple fashion. Under best-first ordering of the lookahead tree, Palphabeta achieves $k^{1/2}$ speedup with k processors. The second distributed algorithm, mwf, orders work to be done by slaves in a more sophisticated manner. Under best-first ordering of a chess lookahead tree, mwf achieves $k^{0.8}$ speedup with k processors. Our work with the parallel alpha-beta algorithms has led to the discovery of an optimization of the serial algorithm. This

optimization, called Palphabeta, is discussed in the appendix.

The question remains whether there exists an optimal parallel algorithm for alpha-beta search. We moved toward optimality in going from Palphabeta to mwf, but abandoned deep cutoffs along the way to simplify the analysis. Any optimal algorithm must achieve these cutoffs. Mw should also be made more practical by increasing the number of processors that can be used to meet a time limit. Currently, mw must run on a processor tree that is less than half the height of the lookahead tree being searched.

Both Palphabeta and Mw assumed that the logical topology of the multicomputer was a tree. This assumption allowed us to write down recursive relations that, when solved, gave the finishing time of the algorithms. Unfortunately, this assumption increased slave idle time: Some of processor X's slaves are idle because X has more slaves than work, while at the same time processor Y has more work than slaves and could use those idle slaves. A distributed algorithm might view the collection of computers as a uniform pool. With this organization, idle time might be reduced. Further research should be directed toward the question of how to organize a pool of processors to perform parallel alpha-beta search.

7.2. PIECEWISE-SERIAL ITERATIVE METHODS

The Jacobi method is an iterative numerical technique for solving certain partial differential equations. We have shown how locally-defined iterative methods give rise to natural multicomputer algorithms. In particular, the grid and the tree algorithms map parts of the problem onto individual processors. Each processor (or terminal processor in the case of a tree multicomputer) engages in serial computation on its region and communicates border values to its neighbors when those values become available.

Our analysis derives the running time of the grid and the tree algorithms with respect to per-message overhead, per-point communication time, and per-point computation time. As long as each machine has a significant amount of work to perform, message passing does not seriously degrade performance. The grid method is more efficient than the tree method, but the semi-synchronous optimization of the tree algorithm is more efficient than the grid algorithm if it is compute-bound. All three algorithms give nearly N-fold speedup with N machines on large problems; the speedup approaches the number of slave processors as the problem size goes to infinity. The efficiency of the tree algorithms depends on the tree fanout; we have shown that the optimal fanout is four. When a tree algorithm is used to solve problems in M dimensions, the optimal fanout is

2^M.

We have shown how to apply the tree algorithms to non-uniform regions both statically and dynamically. These modified algorithms shed load in a natural way from one slave to another when one has a larger area or more expensive computation.

The research reported here can be extended in several ways. We have assumed that the machines that compose the multicomputer are reliable. Experience shows that large assemblages of computers are very likely to have individual malfunctioning elements fairly often. If a machine should fail in such a way that it no longer receives or transmits data, the overall calculation should be able to continue.

Consider the following grid of machines:

```
A B C D
E F G H
I J K L
M N O P
```

Let us suppose that J fails. In the case that each machine only deals with one mesh point, a reasonable adaptation is for machine K to use values from its neighbors, G, L, and O, as before, but to average values from F and N to estimate a value for J. Likewise, machine N will average values from I and K to substitute for missing values from J. However, if each machine is responsible for a sizable patch of mesh points, the recovery strategy is not so clear. Further research should investigate strategies for

continuation when components fail. In the tree algorithm, dynamic region encroachment provides a natural algorithm for continuing when a single processor fails: All of a failed processor's load can be shifted to its siblings.

Dynamic region encroachment presents a method for redistributing subregions as the computation progresses to give each machine a similar amount of work. The question remains how often this redistribution should be done, since it requires a significant movement of data and an interruption of normal computation.

We expect that the advent of large-scale multicomputers will encourage further investigation into parallel algorithms for solving large numerical problems. For these algorithms to be efficient, they must be able to perform relatively large amounts of computation based on relatively small amounts of communication. As we saw in the case of the semi-synchronous tree algorithm, careful attention to the order of computation and communication can reduce communication cost.

7.3. QUOTIENT NETWORKS

By showing how to emulate a large interconnection network with a smaller network of the same topological family, we have presented a method for converting large-network al-

gorithms into more practical small-network algorithms. It is important that the emulation does not in any way depend on the intended computation. Hence we can produce small-network versions of any large-network algorithm. The emulation itself produces no loss of efficiency, but allows us to perform the computation on a range of smaller machines.

7.4. PARALLEL PROGRAMMING PROVERBS

In this section we review some of the lessons learned while designing and implementing distributed algorithms.

7.4.1. Large Computation per Message

Distributed systems, unlike serial systems, must spend part of their time passing messages. Since message-passing in distributed systems can be time consuming, we are interested in algorithms that do as little communication as possible. Each message should invoke, or be a summary of, a large computation. Many of the algorithms presented in this thesis illustrate this principle.

1. Palphabeta and mwf use messages only to invoke or summarize the search of a large subtree. As the size of the lookahead tree increases, the amount of computation increases, but total message-passing time remains constant. Indeed, message-passing time does

not even appear in the speedup formulas for our parallel alpha-beta algorithms, since these formulas represent limiting values as the lookahead tree becomes larger. If instead we had exploited parallelism in move generation or static evaluation, then total message-passing time would have remained proportional to computation time.

2. The grid and tree algorithms for locally-defined iterative calculations were careful to partition the problem grid into segments with maximum area/boundary ratio. Since only boundary points need to be communicated to neighbors at each timestep, communication time is minimized. The area/boundary ratio increases quadratically with the size of the problem grid.

Hence the communication time does not appear in speedup figures for large problem grids.

7.4.2. Do Interesting Work First

The semi-synchronous algorithms for locally-defined iterative calculations calculate new boundary points before new interior points. As soon as the new boundary points were available, they are sent to the master processor for distribution to other slaves. Only then does the calculation of interior points begin. By giving a head start to the dissemination of this "interesting information", the

computation is speeded up, since the processor that is interested in the information is not kept waiting.

7.4.3. Do Mandatory Work First

The alpha-beta pruning technique prunes away some parts of the lookahead tree on the basis of information originating in other parts of the tree. Both of our parallel alpha-beta algorithms, Palphabeta and Mwf, achieve parallelism by dividing the lookahead tree into subtrees and assigning each subtree to a separate task. Hence the results of some tasks can cause other tasks to be canceled. Suppose that task A cannot be canceled and that its result might cancel task B. Palphabeta is likely to lose the possible savings by executing A and B concurrently. Mwf achieves the savings, if possible, by executing A concurrently with some other non-cancelable task. Mwf's smart behavior results in an improvement in speedup for searching best-first chess lookahead trees: With P processors Mwf achieves $p^{0.8}$ speedup, as compared with $p^{0.5}$ speedup for Palphabeta.

7.4.4. Do Something

Into the life of every processor must come some idle time. If idleness is likely to happen often, we should arrange for something useful for the processor to do. For

example, in the parallel alpha-beta algorithms, it can happen that

1. The queue of tasks is empty.
2. Some slaves are still busy with tasks.
3. Some slaves are idle.

We would like to give the idle slaves something useful to do, but the queue is empty. Worse, we cannot refill the queue until all the busy slaves finish. But with a little imagination we can find work: For example, we can assign the same subtrees that are still being worked on, but with smaller windows. Hence the same subtree might be searched concurrently by more than processor, but with different-sized windows. If a processor with one of the smaller windows finishes first, then our strategy has paid off; we can send an alpha-beta update to the other processors working on the same tree. These processors then speed up or even terminate.

Appendix A - Some Optimizations of α - β Search

In this appendix we propose three optimizations of the serial α - β algorithm.

A.1. FALPHABETA

The first optimization, called Falphabeta for "fail-soft alpha-beta search", is completely riskless in the sense that it never searches more nodes than alphabeta. Although it requires a slight constant overhead, it results in a slight expected speedup whenever an initial window other than $(-\infty, +\infty)$ is used. Here is Falphabeta:

```

1 function Falphabeta(p: position;  $\alpha, \beta$ : integer): integer;
2 begin integer m, i, t, d;
3   determine the successor positions  $p_1, \dots, p_d$ ;
4   if  $d = 0$  then return(staticvalue(p));
5   m :=  $-\infty$ ;
6   for i := 1 to d do
7     begin
8       t := -Falphabeta( $p_i, -\beta, -\max(m, d)$ );
9       if  $t > m$  then m := t;
10      if  $m \geq \beta$  then return(m);
11    end;
12  return(m);
13 end;
```

Falphabeta differs from alphabeta only in that m has been initialized to $-\infty$ instead of α . In order to keep this change from affecting the third actual parameter to

the recursive call to Falphabeta (line 8), " $-\infty$ " is changed to " $-\max(m, d)$ ". The computational overhead of repeatedly computing the maximum of m and d is the only added expense of Falphabeta. As mentioned in Chapter 4, the value returned by the call to the original α - β procedure, $\text{alphabeta}(p, \alpha, \beta)$, obeys the following relation with respect to the true negamax value of a search tree:

If $\text{alphabeta} \leq d$, then $\text{negamax}(p) \leq d$,
 if $\text{alphabeta} \geq \beta$, then $\text{negamax}(p) \geq \beta$,
 if $d < \text{alphabeta} < \beta$ then $\text{negamax}(p) = \text{alphabeta}$.

Falphabeta obeys a stronger relation:

Theorem A. If p is the root node of a lookahead tree, and if α and β are integers satisfying $\alpha < \beta$, then the value Falphabeta returned by $\text{Falphabeta}(p, \alpha, \beta)$ satisfies:

If $\text{Falphabeta} \leq \alpha$, then $\text{negamax}(p) \leq \text{Falphabeta}$,
 if $\text{Falphabeta} \geq \beta$, then $\text{negamax}(p) \geq \text{Falphabeta}$,
 if $\alpha < \text{Falphabeta} < \beta$ then $\text{negamax}(p) = \text{Falphabeta}$.

Proof. The relations clearly hold if p is a terminal node.

Assume for the induction step that the relations hold for any tree of height k or less. Let p be the root of a tree of height k + 1. Let $p_1, \dots, p_d$ be the successors of p. Each p_i is the root of a tree of height k or less.

1) If

$\text{Falphabeta}(p, \alpha, \beta) \leq \alpha$,

then for all $1 \leq i \leq d$, we have

$\text{Falphabeta}(p_i, -\beta, -\alpha) \geq -\alpha$.

By the induction hypothesis, we have

$$\text{negamax}(p_1) \geq \text{Falphabeta}(p_1, -\beta, -\alpha).$$

Hence

$$\max_1 \text{negamax}(p_1) \leq \max_1 -\text{Falphabeta}(p_1, -\beta, -\alpha).$$

Hence $\text{negamax}(p) \leq \text{Falphabeta}(p, \alpha, \beta)$.

2) If $\text{Falphabeta}(p, \alpha, \beta) \geq \beta$, then there exists i such that

$$-\text{Falphabeta}(p_1, -\beta, -\alpha') = \text{Falphabeta}(p, \alpha, \beta) \geq \beta,$$

for some α' such that $\alpha \leq \alpha'$. By the induction hypothesis, we may conclude that

$$\text{negamax}(p_1) \leq \text{Falphabeta}(p_1, -\beta, -\alpha').$$

Hence $\text{negamax}(p) = \max_1 \text{negamax}(p_1) \geq \text{Falphabeta}(p, \alpha, \beta)$.

3) If $\alpha < \text{Falphabeta}(p, \alpha, \beta) < \beta$, then let i be the smallest integer such that

$$-\text{Falphabeta}(p_1, -\beta, -\alpha') = \text{Falphabeta}(p, \alpha, \beta),$$

for some α' such that $\text{Falphabeta}(p, \alpha, \beta) > \alpha' \geq \alpha$. Hence

$$-\beta < \text{Falphabeta}(p_1, -\beta, -\alpha') < -\alpha'.$$

Therefore, by the induction hypothesis,

$$\begin{aligned} \text{negamax}(p_1) \\ = \text{Falphabeta}(p_1, -\beta, -\alpha') = -\text{Falphabeta}(p, \alpha, \beta). \end{aligned}$$

Since $\text{negamax}(p) = \max_1 \text{negamax}(p_1)$, we have

$$\text{negamax}(p) = \text{Falphabeta}(p, \alpha, \beta);$$

Q.E.D.

Theorem A implies that Falphabeta can give a tighter bound than alphabeta on the true value of the tree when it fails high or low. Falphabeta "fails softer" than alphabe-

ta. The extra information that Falphabeta gives can be used in two ways. First, this information is useful whenever the common wisdom "start with a tight window" is followed. If the tight window (α, β) causes the search to fail, the penalty of doing the entire search over again must be paid. With normal α - β search, this second search must be done with the window $(-\infty, \alpha)$ (if the original search failed low) or $(\beta, +\infty)$ (if the original search failed high). Falphabeta reduces this penalty: A low fail will sometimes return a number $k < \alpha$, and the second search can be started with the tighter window $(-\infty, k)$. We can expect a similar saving when a high fail occurs.

We need two definitions to explain the second use of Falphabeta. Staged iteration evaluates a lookahead tree to depth N by first searching to depths 2, 3, ..., $N-1$. After each stage, the principal continuation (the path the game would take if each player played optimally) is saved. The next stage begins its depth-first search by descending to the end of this path; whenever a node on the principal continuation is visited, its principal child is examined first. Staged iteration provides very reliable best-first move ordering at type-one nodes so it actually decreases the number of nodes examined in programs.

Forward pruning is opposed to α - β pruning, which is a form of backward pruning, cuts off a node of a tree before

that P_d is the best move, since its negamax value has been established to be lower than any other. The modified algorithm does not discover the value of the best move when that move is evaluated last. However, it still determines which move is best. This slight reduction in information can buy a time savings, since the evaluation of P_d has a very narrow window.

A parallel version of this technique was discussed in subsection 4.4.3. under the name "alpha-raising". The new algorithm will be called Lalphabeta, short for "last-move-with-minimal-window alpha-beta search".

```

1 function Lalphabeta (p: position; d,  $\beta$ : integer): integer;
2 begin integer m, i, t, d;
3   determine the successor positions  $P_1, \dots, P_d$ ;
4   if  $d = 0$  then return (staticvalue(p));
5    $m := d$ ;
6   for i := 1 to d-1 do
7     begin
8       t := -alphabeta( $P_i, -\beta, -m$ );
9       if  $t > m$  then  $m := t$ ;
10      if  $m \geq \beta$  then return(m);
11    end;
12  t := -alphabeta( $P_d, -m-1, -m$ );
13  if  $t > m$  then  $m := t$ ;
14  return(m);
15 end;
```

Lalphabeta provides an elegant solution to the forced-move problem: Programmers writing their first game-playing program often find to their amusement that al-

fully investigating any of its siblings. It is obvious that forward pruning can provide enormous savings in tree search. Unfortunately, forward pruning is very risky. No one has yet discovered how to perform forward pruning without occasionally pruning away the best move. (The very best chess programs do not perform forward pruning.) One of the reasons that forward pruning has not been successfully implemented is that when a poor move is evaluated after a better move, alphabeta assigns both the same score (except when the poor move is within two moves of the terminal node that produces the poor score). Lalphabeta sometimes gives the poor move a more appropriate value, so it may provide a basis for reliably pruning the move during the next stage of a staged iteration.

A.2. LALPHABETA

When alphabeta is recursively called on the last successor P_d of the root of the entire tree, p , the current value $-\beta$ ($-\infty$) is passed as formal parameter d . Suppose that $-m-1$ is passed instead. If P_d is not the best move, then $\text{negamax}(P_d) \geq -m$, and $\text{alphabeta}(P_d, -m-1, -m)$ falls high as before. If P_d is the best move, then $\text{negamax}(P_d) \leq -m-1$, and so $\text{alphabeta}(P_d, -m-1, -m)$ falls low instead of succeeding. Nevertheless, the algorithm can still conclude

phabeta conducts a full-scale search even though only one move is available to the computer. Lalphabeta searches the one available move with the window $(\infty - 1, \infty)$. Besides greatly speeding up the search, Lalphabeta actually performs useful work in this case: It decides if it should resign!

A.3. CALPHABETA

The third optimization, called Calphabeta because it is called only on nodes along the principal continuation, is a generalization of Lalphabeta, and profits from Falphabeta, but carries with it the risk that in certain cases more nodes will be examined.

```

1 function Calphabeta(p: position): integer;
2 begin integer m, i, t, d;
3   generate the successors  $P_1, \dots, P_d$ ;
4   if  $d = 0$  then return(staticvalue(p));
5    $m = -\text{Calphabeta}(P_1)$ ;
6   for  $i := 2$  to  $d$  do
7     begin
8        $t = -\text{Falphabeta}(P_i, -m-1, -m)$ ;
9       if  $t > m$  then  $m := -\text{Falphabeta}(P_i, -\infty, -t)$ 
10    end;
11  return(m);
12 end;
```

If Calphabeta evaluates the best move first at type one nodes, then all of the other subtrees are searched with

a minimal window. On the other hand, every subtree that is better than its older siblings must be searched twice, resulting in more work. The first search, conducted with the minimal window, discovers that the subtree is the new best one, and really should not have been searched with the minimal window after all. The second search discovers the true value. It is important that the best move be evaluated first with high enough probability that the savings outweigh the penalties. Staged iteration can generate the best move first with high probability. If the principal line established for the $(N-1)$ th stage is a prefix of the principal line for the N th stage, then at the N th stage virtually the entire tree is searched with a minimal window.

A.4. MEASUREMENTS

To measure the improvement due to Lalphabeta and Calphabeta, four checkers games were played, during which the program made 46 moves. Each move selection was repeated six times, one for each of the six algorithms: alphabeta, Lalphabeta, Calphabeta, salphabeta, sLalphabeta, and sCalphabeta. Alphabeta, Lalphabeta, and Calphabeta have already been defined, and were done without staging. Salphabeta, sLalphabeta, and sCalphabeta are the staged ver-

sions of these three algorithms. During each of the 46*6 move selections, the number of nodes visited was counted, providing 46 values for alphabeta, Lalphabeta, Calphabeta, salphabeta, sLalphabeta, and sCalphabeta, and hence 46 values for the five derived quantities alphabeta/salphabeta, Lalphabeta/alphabeta, Calphabeta/alphabeta, sLalphabeta/salphabeta, and sCalphabeta/salphabeta.

Table 1 shows statistics for alphabeta/salphabeta.

Checkers, unlike chess, does not profit from staging, possibly due to checker's smaller branching factor. On the average, alphabeta searched only 81% as many nodes as salphabeta.

Table 1: alphabeta/salphabeta

Minimum	0.019
Maximum	2.768
Average	0.808
Standard Deviation	0.462

Table 2 gives statistics for Lalphabeta/alphabeta, Calphabeta/alphabeta, sLalphabeta/salphabeta, and sCalphabeta/salphabeta.

Table 2:

	Lalphabeta/alphabeta	Calphabeta/alphabeta
Minimum	0.881	0.666
Maximum	1.000	5.750
Average	0.987	1.163
Standard Deviation	0.024	0.868

	sLalphabeta/salphabeta	sCalphabeta/salphabeta
Minimum	0.899	0.696
Maximum	1.000	2.174
Average	0.988	0.960
Standard Deviation	0.023	0.227

As expected, staged iteration was crucial to making Calphabeta work at all; without staging, Calphabeta actually searched more nodes than alphabeta. However, the measurements of sCalphabeta (Calphabeta with staging) are disappointing. SCalphabeta searched only four percent fewer nodes than salphabeta. Since savings from starting with a narrow window (an optimization that could be used in place of Calphabeta or sCalphabeta) are on the order of 20 percent [9], Calphabeta and sCalphabeta are probably not to be recommended.

Lalphabeta and sLalphabeta, on the other hand, are unqualified (albeit small) successes. On the average, each searches about one percent fewer nodes than the corresponding standard algorithm. Although this improvement is not great, the optimization is clearly a good bargain, since its space overhead is insignificant and its time overhead is zero. Lalphabeta is never slower than alphabeta and sLalphabeta is never slower than salphabeta. Therefore, every game-playing program that uses $d-p$ search should use some form of Lalphabeta.

REFERENCES

- [1] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," Communications of the ACM **21**, 8, pp. 613-641 (August 1978).
- [2] W. J. Bouknight et al., "The Illiac IV system," Proc. IEEE **60**, 4, pp. 369-388 (April 1972).
- [3] W. A. Wulf and C. G. Bell, "C.mmp -- a multi-mini-processor," Proc. AFIPS 1972 Fall Joint Computer Conference **41**, Part II, pp. 765-777 (1972).
- [4] J. A. Rudolph, "A production implementation of an associative array processor - Staran," AFIPS Fall 1972 **41**, AFIPS Press, pp. 229-241 (1972).
- [5] A. J. Evensen and J. L. Troy, "Introduction to the architecture of a 288 element PEPE," Proc. 1973 Sagamore Conference on Parallel Processing, (August 1973).
- [6] P. M. Flanders, D. J. Hunt, S. F. Reddaway, and D. Parkinson, "Efficient high speed computing with the Distributed Array Processor," Symposium on High Speed Computer and Algorithm Organization, pp. 113-128 (1977).
- [7] M. H. Solomon and R. A. Finkel, "The Roscoe distributed operating system," Proc. 7th Symposium on Operating Systems Principles, pp. 108-114 (December 1979).
- [8] M. J. Flynn, "Very high-speed computing systems," Proceedings of the IEEE **54**, 12, pp. 1901-1909 (December 1966).
- [9] G. M. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors," Department of Computer Science, Carnegie-Mellon University (April 1978).
- [10] M. C. Pease, "An adaptation of the fast Fourier Transform for parallel processing," Journal of the ACM **15**, 2, pp. 252-264 (April 1968).
- [11] K. E. Batcher, "Sorting networks and their applications," Proc. Spring Joint Comput. Conf. **32**, pp. 307-314 (1968).

- [12] L. Csanky, "Fast parallel matrix inversion algorithm," SIAM J. Computing **5**, 4, pp. 618-623 (December 1976).
- [13] H. S. Stone, "Parallel processing with the perfect shuffle," IEEE Transactions on Computers **C-20**, 2, pp. 153-161 (February 1971).
- [14] D. E. Muller and F. P. Preparata, "Bounds to complexities of networks for sorting and for switching," Journal of the ACM **22**, 2, pp. 195-201 (April 1975).
- [15] D. E. Knuth, The Art of Computer Programming Volume 3--Sorting and Searching, Addison-Wesley (1973).
- [16] S. Even, "Parallelism in tape-sorting," Communications of the ACM **17**, 4, pp. 202-204 (April 1974).
- [17] L. G. Valiant, "Parallelism in Comparison Problems," SIAM Journal of Computing **4**, 3, pp. 348-355 (September 1975).
- [18] F. Gavril, "Merging with parallel processors," Communications of the ACM **18**, 10, pp. 588-591 (October 1975).
- [19] D. S. Hirschberg, "Fast Parallel Sorting Algorithms," Communications of the ACM **21**, 8, pp. 657-661 (August 1978).
- [20] F. P. Preparata, "New parallel-sorting schemes," IEEE Transactions on Computers **C-27**, 7, pp. 669-673 (July 1978).
- [21] G. M. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," IEEE Transactions on Computers **C-27**, 1, pp. 84-87 (January 1978).
- [22] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected parallel computer," Communications of the ACM **20**, 4, pp. 263-271 (April 1977).
- [23] H. S. Stone, "Sorting on STAR," IEEE Transactions on Software Engineering **SE-4**, 2, pp. 138-146 (March 1978).
- [24] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Math. Comput. **19**, pp. 297-301 (April 1965).

- [25] J. M. Lemme and J. R. Rice, "Speedup in parallel algorithms for adaptive quadrature," Journal of the ACM 26, 1, pp. 65-71 (January 1979).
- [26] J. F. Traub, "Iterative solution of tridiagonal systems on parallel or vector computers," Complexity of sequential and parallel numerical algorithms, Academic Press, (1973).
- [27] H. S. Stone, "Parallel Tridiagonal Equation Solvers," ACM Transactions on Mathematical Software 1, 4, pp. 289-307 (December 1975).
- [28] H. S. Stone, "Problems of Parallel Computation," Complexity of Sequential and Parallel Numerical Algorithms, Academic Press, (1973).
- [29] S. C. Chen, D. J. Kuck, and A. H. Sameh, "Practical Parallel Band Triangular System Solvers," ACM Transactions on Mathematical Software 4, 3, pp. 270-277 (September 1978).
- [30] W. M. Gentleman, "Some complexity results for matrix computations on parallel processors," Journal of the ACM 25, 1, pp. 112-115 (January 1978).
- [31] F. P. Preparata and D. V. Sarwate, "An improved parallel processor bound in fast matrix inversion," Information Processing Letters 7, 3, pp. 148-150 (April 1978).
- [32] R. A. Finkel, M. H. Solomon, and M. L. Horowitz, "Distributed algorithms for global structuring," Proc. National Computer Conference 48, AFIPS Press, pp. 455-460 (June 1979).
- [33] E. Chang, "An introduction to echo algorithms," Proc. 1st International Conference on Distributed Computers, pp. 193-198 (October 1979).
- [34] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing connected components on parallel computers," Communications of the ACM 22, 8, pp. 461-464 (August 1979).
- [35] C. D. Savage, Parallel algorithms for graph theoretic problems, Computer Science Department, U. of Illinois, Urbana, Ill. (August 1977) Ph.D. Thesis.

- [36] H. J. Berliner, "A chronology of computer chess and its literature," Artificial Intelligence 10, pp. 201-214 (April 1978).
- [37] S. G. Akl, D. T. Barnard, and R. J. Doran, "Simulation and analysis in deriving time and storage requirements for a parallel alpha-beta algorithm," Proc. 1980 International Conference on Parallel Processing, pp. 231-234 (August 1980).
- [38] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," Artificial Intelligence 6, 4, pp. 293-326 (Winter 1975).
- [39] A. L. Samuel, "Some studies in machine learning using the game of checkers, II - recent progress," IBM Journal of Research and Development, pp. 601-617 (November 1967).
- [40] R. Nevanlinna and V. Paatero, Introduction to Complex Analysis, Addison-Wesley (1969).
- [41] Oskar Perron, "Zur Theorie der Matrizes," Math. Ann. 64, pp. 248-263 (1907).
- [42] P. Brinch Hansen, Operating System Principles, Prentice-Hall (1973).
- [43] D. M. Young, Iterative Solution of Large Linear Systems, Academic Press (1971).
- [44] J. L. Rosenfeld, "A case study in programming for parallel-processors," CACM 12, 12, pp. 645-655 (December 1969).
- [45] C. F. R. Weiman and C. E. Grosch, "Parallel processing research in computer science: Relevance to the design of a Navier-Stokes computer," Proc. 1977 International Conference on Parallel Processing, pp. 175-182 (August 1977).
- [46] H. O. Welch, "Numerical weather prediction in the Peper parallel processor," Proc. 1977 International Conference on Parallel Processing, pp. 186-192 (August 1977).
- [47] D. Chazan and W. Miranker, "Chaotic relaxation," Linear Algebra and Appl. 2, pp. 199-222 (1969).

- [48] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," Journal of the ACM 25, 2, pp. 226-244 (April 1978).
- [49] J. L. Bentley and H. T. Kung, "A tree machine for searching problems," Proc. 1979 International Conference on Parallel Processing, pp. 257-266 (August 1979).
- [50] L. J. Siegel, P. T. Mueller, and H. J. Siegel, "FFT algorithms for SIMD machines," Proc. Allerton Conference on Communication, Control, and Computing, pp. 1006-1015 (October 1979).
- [51] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," IEEE Transactions on Computers C-26, 2, pp. 153-161 (February 1977).

APPENDIX B

**Density and Reliability of
Interconnection Topologies for Multicomputers (Ph.D. Thesis)**

Well E. Leland

Technical Report 478
Computer Sciences Department
University of Wisconsin-Madison
July 1982

DENSITY AND RELIABILITY OF
INTERCONNECTION TOPOLOGIES FOR MULTICOMPUTERS

by

WILL EDWARD LELAND

A thesis submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

MAY 1982

© Copyright by Will Edward Leland 1982

All rights reserved

Abstract

In a multicomputer consisting of processors that communicate over point-to-point full-duplex communication lines, the distance between two processors is the number of message forwarding steps required for them to communicate. The diameter of a network is the longest distance between two processors, and the degree is the largest number of lines from any one processor. We examine the problem of minimizing the diameter k of a network as a function of the number N of processors, for a fixed degree d .

We define a new general method, the double-exchange topology, that produces families of interconnection topologies that are denser than any families previously proposed. In particular, we have discovered families of networks with $d = 3$, $k = 1.47 \lg N$; $d = 4$, $k = 0.947 \lg N$; and $d = 5$, $k = .75 \lg N$. The best previously-known results were $d = 3$, $k = 2 \lg N$; $d = 4$, $k = \lg N$; and $d = 5$, $k = .77 \lg N$.

These new families have a regular structure that admits efficient routing algorithms. They have a close relationship with previously-proposed (and less dense) families, allowing us to exploit known results on algorithm-mapping and VLSI layout.

Finally, we explore the issue of reliability. An intricate proof shows that our degree-3 families are 3-connected (the best possible connectivity for degree-3 networks) -- that is, at least three node failures are required to partition the network. We then propose two new measures of network reliability, called local reroutability and average random failset size, that are particularly appropriate for evaluating multicomputer topologies. Both by the standard of connectivity and by our proposed measures, our double-exchange families of interconnection topologies are highly reliable as well as exceptionally dense.

Acknowledgements

Professor Marvin Solomon has been a remarkable advisor: patient, encouraging, and always willing to take the time to think seriously about my research, to read carefully my interminable revisions, and to improve both beyond measure. The thoughtful comments and helpful conversations of Professors Raphael Finkel, S. Diane Smith, and Jim Goodman did much to clarify my writing and my reasoning; I am indebted to them and to Professor Charles Kime for their efforts with my thesis.

This thesis would never have happened without Professor Leonard Uhr, whose enthusiasm introduced me to the question of density, and whose interest and results encouraged me to pursue it. The expertise of Professors Sam Bent and E. F. Moore saved me much work, while the generosity of Sheryl Pomeroy made the job of producing the finished thesis far less miserable than I had expected. I am grateful to my parents, my brother, and the Department of Computer Sciences for their undeserved patience and support. Bill Cox and Bill Wickart have been good friends and colleagues for years; their friendship, and the friendship of the Cross Trails Square Dance Club, the Friends of the Arboretum, the Arachne research group, and many others made a stressful time almost pleasant.

I cannot offer adequate words to thank my wife, Mary, for her hours with my thesis; I can offer my love and a promise to reciprocate for her thesis.

This research was supported in part by the Defense Advanced Projects Research Agency under Navy contract N00014-81-C-2151.

Table of Contents

Abstract.	iii
Acknowledgements.	v
Table of Contents.	vii
List of Figures.	xii
List of Tables.	xiii
Chapter 1. Introduction.	1
1.1. Introduction.	1
1.2. Overview.	3
Chapter 2. Dense Multicomputer Graphs.	8
2.1. Definitions.	9
2.2. Graph Product Families.	15
2.2.1. The complete hypercube.	16
2.2.2. The mesh-connected hypercube.	17

2.3. Tree Variations.	19
2.3.1. Multi-tree Structures.	19
2.3.2. The hypertree family.	20
2.4. Single-Exchange Graphs.	23
2.4.1. Definitions.	23
2.4.2. The diameter of a single-exchange graph.	25
2.4.3. Routing in single-exchange families.	29
2.4.4. Specific single-exchange families.	31
2.4.4.1. Cube-connected-cycles.	31
2.4.4.2. The lens.	33
2.4.4.3. The shuffle-exchange graph.	34
2.4.4.4. The de Bruijn networks.	35
2.4.4.5. The C_g graph.	37
2.4.5. Operations that construct single-exchange families.	38
2.5. Double-Exchange Graphs.	42
2.5.1. Definitions.	42
2.5.2. The diameter of a double-exchange set.	43
2.5.3. Routing in double-exchange families.	48

2.5.4. Specific double-exchange families.	49	4.1.1.2. Tree variations.	79
2.5.4.1. Degree 3.	49	4.1.1.3. Single-exchange graphs	80
2.5.4.2. Degree 4.	51	4.1.2. Connectivity of the Moebius graph	90
2.5.4.3. Degree 5.	52	4.1.3. Connectivity of the shuffle-exchange.	100
2.5.5. Variations of the double-exchange.	52	4.1.4. Connectivity of the even double-exchange.	101
2.5.5.1. The twisted double-exchange	53	4.2. Local Retrouting.	103
2.5.5.2. The Moebius graph	54	4.2.1. Definitions	103
2.5.5.3. The multistage double-exchange.	54	4.2.2. Routability of specific families.	106
2.5.6. Interpolating double-exchange families	56	4.2.2.1. The hypercube.	107
2.6. Summary	59	4.2.2.2. Tree variations.	108
Chapter 3. Algorithms and Layout.	61	4.2.2.3. The cube-connected-cycles.	111
3.1. Properties of the Homomorphism.	62	4.2.2.4. The shuffle-exchange	114
3.2. VLSI Layout of the Even Double-Exchange	65	4.2.2.5. The lens	115
3.3. Summary	75	4.2.2.6. The de Bruijn and C_g^* graphs.	119
Chapter 4. Reliability.	76	4.2.2.7. The elided Moebius	122
4.1. Connectivity.	78	4.2.2.8. The elided even double-exchange.	123
4.1.1. Previous multicomputer graphs	78	4.2.2.9. Summary.	124
4.1.1.1. Graph product families	78	4.3. Average Random Fallsets.	127
		4.3.1. Definitions and basic results	128
		4.3.2. An analytic lower bound	136

4.4. Summary	142
Chapter 5. Conclusions	143
5.1. Summary	143
5.2. Future Research	145
5.3. Conclusions	148
References	149

List of Figures

2.1 The Petersen Graph	11
3.1 Mapping of the Shuffle-Exchange to the Complex Plane, for $n = 5$	69
3.2 Layout of the Shuffle-Exchange, for $n = 5$	71
3.2 Layout of the Even Double-Exchange, for $n = 5$	74
4.1 Rerouting for the Hypertree	110
4.2 Vertex Rerouting for the Cube-Connected-Cycles	113
4.3 Vertex Rerouting for the Lens	118

Chapter 1
Introduction

List of Tables

2.1 Degrees and Costs of Graph Families 60

4.1 Inner Paths for Lemma 1.1.4. 90

4.2 Paths for Lemma 1.2.2. 96

4.3 Region Sizes and Reroute Costs
for Graph Families 126

4.4 ARP Simulation Results 131

4.5 Comparison of ARP Simulation
and Analytic Bound 140

1.1. Introduction

The rapid development of microprocessors and VLSI makes the concept of a multicomputer increasingly attractive for achieving high effective processing rates. A multicomputer consists of many cooperating independent asynchronous processors; in a multicomputer, the processors must have reliable high-bandwidth communication facilities. Among the various possible interconnection designs, including busses and switching networks, point-to-point dedicated interconnections are very well suited to the VLSI requirements of restricted off-chip bandwidth and high logic-to-pin ratios. In particular, this research studies interconnection topologies for multicomputers in which each processor communicates with other processors by sending and forwarding messages over point-to-point full-duplex communication lines.

The designer of such multicomputers needs to discover interconnection topologies for the number of processors and of I/O ports per processor available, and to compare their effectiveness. Ad hoc design is adequate for small multicomputers, but has serious drawbacks when we consider scores or hundreds of processors. For each

new set of constraints, we must generate a topology; once we have found one, we must solve the problems of addressing and routing on it (even in the face of possible failures), find algorithms that exploit its particular interconnections, analyze its reliability, and compare it with other candidate topologies.

The designer is rescued from these difficulties by the formal definition of families of possible interconnections. The problems of addressing, routing, algorithm design, and reliability can then be solved for all members of a family at once, and entire families compared. We have discovered families of interconnections that are denser than any previously proposed. These interconnection topologies are also highly regular and resistant to isolated failures.

We investigate the question of fault tolerance in some detail, proposing new measures of reliability and developing mathematical tools for studying these measures.

1.2. Overview

Graph theory provides a convenient formal model for these interconnection topologies. The vertices (nodes, points) of a graph represent the processors and the edges (lines, arcs) represent the communication lines. The same formal model is used for switching networks (by interpreting some vertices as switches and others as processors or memories), and for computer communications networks (by interpreting the vertices as geographically remote hosts). To emphasize the distinction between our interpretation and other multiprocessor models that employ graph theory, we call our abstract models multicomputer graphs.

The different interpretations imply different criteria for evaluating graphs. Unlike switching networks, multicomputer graphs can ignore switch-setting algorithms and considerations of blocking; however, since the processors have limited total I/O bandwidth, our graphs must have very small degree (that is, very few edges per vertex). We concentrate on degrees 3, 4, and 5, with some consideration of general families that extend to higher degrees. Unlike computer communications networks, multicomputer graphs do not need to minimize the total number of edges or vertices involved, and cannot afford large ad hoc routing tables. We therefore

the diameter-degree product criterion, the degree-5 double-exchange family is, in fact, the densest family known at any degree.

Chapter 3 concentrates on a particular variety of double-exchange graph, exploring the problems of layout and of mapping algorithms to processors. A simple homomorphism from the well-studied shuffle-exchange family onto this double-exchange allows a computationally uniform emulation of any shuffle-exchange algorithm, and, by exploiting known results for the shuffle-exchange, provides both an asymptotic lower bound for the area required to lay out the new family and a good layout for it.

Chapter 4 returns to comparing all multicomputer graph families, using measures of reliability. Connectivity, the minimum number of failures required to disconnect a graph, is a measure of reliability often used for computer networks. After summarizing the known connectivities for previous multicomputer graph families, we show that three dense degree-3 families -- the well-known shuffle-exchange and two of our much denser double-exchange families -- can be made 3-connected by minor changes that preserve their degree, density, routing, and interpolability. No higher connectivity is possible for degree-3 graphs. We then propose two meas-

require graphs with address assignments that permit simple routing algorithms. This requirement is met by studying uniform infinite families of graphs, in which all members can use the same routing algorithm regardless of N , the number of processors.

A dense graph is one whose diameter (the greatest distance between any two vertices) is low for its degree and number of vertices; there is a limit to how small the diameter of any graph may be, given N and its degree. A dense topology is important not only for reduced communication delay and congestion, but also for enhanced reliability. Chapter 2 compares known dense multicomputer graph families using the cost measures of diameter and diameter times degree, as well as considerations of routing and interpolability (the ability to produce designs for any given number of processors). While reviewing previously-proposed multicomputer graph families in terms of these criteria, it presents a general construction -- the single-exchange family -- that includes most of the densest families proposed, and provides linear interpolation and a uniform routing algorithm for them. Finally, it introduces a new construction -- the double-exchange family -- that produces the densest known graph families at degrees 3, 4, and 5, as well as having simple routing and interpolation. Under

ures of reliability that are suitable for the distributed nature of the multicomputer: local reroutability and Average Random Failset size.

An infinite family of graphs is locally reroutable if there is some fixed number r , independent of N , such that any given isolated failure requires that at most r vertices modify their routing algorithms to detour messages around the fault. We measure the cost of this local rerouting by the minimal r that suffices for all members of a family, and by the increase in the length of the detoured paths.

The Average Random Failset (ARF) size of a graph is the expected number of random failures required to partition it; unlike most probabilistic measures of network reliability, it is independent of the fault probabilities of the individual elements. After determining the costs of local rerouting in the multicomputer graph families, Chapter 4 compares their ARF sizes by simulation and derives a lower bound on ARF size based on local reroutability.

The multicomputer designer can therefore use our research both for reliability indices that are suitable for comparing multicomputer topologies, and for specific families of topologies that have high density, natural address assignments, simple routing algorithms, uniform

rerouting methods for bypassing failed processors or communications lines, and high reliability.

Chapter 2

Dense Multicomputer Graphs

This chapter reviews families of graphs previously proposed for multicomputer interconnection, compares them according to several standard cost criteria, and develops a new construction that produces several cheaper families. To emphasize the distinction between our model, in which all vertices represent independent processors communicating via messages over full-duplex links (represented by edges), and other multiprocessor models that employ graph theory (such as permutation networks, in which most of the vertices represent switches in a network that provides interprocessor communication), we will call our abstract versions multicomputer graphs. The reader should be aware that there are terminological conflicts between several closely related research areas; for an example that will arise later in this chapter, the various permutation networks called "shuffle-exchange networks" [1] are not exactly the same graphs as the "shuffle-exchange graphs" in the multicomputer graph literature [2,3,4].

2.1. Definitions

$\lg$ denotes the base two logarithm. $[x]$ denotes the largest integer not greater than x ; $\lceil x \rceil$ denotes the smallest integer not less than x ; and $i \text{ div } j$ denotes $\lfloor i/j \rfloor$. We use b^n to denote the set of b -ary strings of length n . Whenever we use integers drawn from the set $\{0, 1, \dots, c-1\}$ for any c , we employ the convention that our arithmetic is modulo c ; for example, given $x \in b^n$, all arithmetic involving digits of x is modulo b , while arithmetic involving subscripts of its digits is modulo n . If x is a string of digits, x^* denotes a string formed by repeating x some number of times; the number of repetitions will be clear from context.

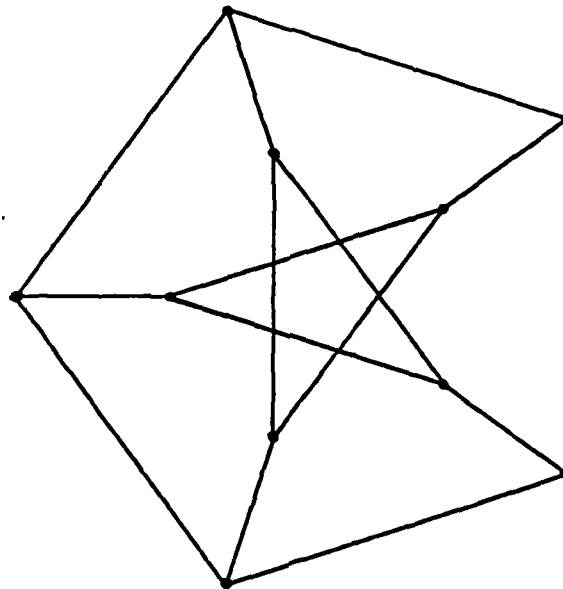
A graph consists of a nonempty set V of vertices and a set E of edges, or pairs of adjacent vertices. N will denote the number of vertices, and will sometimes be referred to as the size of a graph. All graphs will be simple, connected, and undirected. A simple graph has no self-loops or multiple edges; a connected graph has a path between any two vertices; an undirected graph has all edges undirected; if vertex v is adjacent to vertex w , then w is adjacent to v . The distance between two vertices is the length of a shortest path between them. The diameter, k , of a graph is the maximum distance between any two of its vertices. The degree of a

vertex is the number of edges incident on it; the degree of a graph, denoted d , is the maximum of the degrees of its vertices. A graph of degree three is called trivalent. If all vertices have the same degree, the graph is regular. (See Harary [5] for graph terminology not defined here.)

Two families of graphs that will be used heavily in later constructions have standard names in graph theory. C_N denotes a cycle (or ring) of N vertices, each connected to two neighbors. K_N denotes a complete graph with N vertices, each connected to all the others to give a diameter of 1. Another graph that appears often is the Petersen graph (Figure 2.1) which has degree 3, diameter 2, and 10 vertices.

Two quantitative cost criteria will be used in this chapter to compare the graph families as multicomputer graphs: diameter and diameter $\cdot$ degree. The latter criterion will be called the product cost of a graph. Graphs that have low diameter and product costs are called dense, and are extremely important for practical interconnection topologies: a network's diameter provides a lower bound on the message steps required by broadcast or echo algorithms [6] and an upper bound to the routing distance between any two vertices. Diameter also directly affects traffic congestion and several

Figure 2.1



The Petersen Graph

measures of network reliability [7,8,9].

The diameter that can be achieved for any given number of vertices, however, depends strongly on the degree of the graph. A theoretical limit is given by the Moore bound [10]. For degree 2, the best possible graphs are cycles, C_N , with N odd and diameter $k = \lfloor N/2 \rfloor$. For degree $d > 2$, $N \leq (d(d-1)^{k-2}) / (d-2)$, so $k \geq \lg(N) / \lg(d-1) + O(1)$. This bound is derived by enumerating the vertices in the tree created by taking any vertex for the root and treating its d neighbors as the roots of $(d-1)$ -ary subtrees of height $k-1$. Clearly, this tree includes every vertex that could lie within distance k of its root. The Moore bound is provably unreachably for all diameters > 2 , and, at diameter 2, for all degrees except 2, 3, 7, and possibly 57 [11,12]. This bound has not been refined, except for diameter 2, where Erdős [13] proved that, except for degrees 2, 3, 7, and possibly 57, all graphs have $N < d^2$.

The product cost, $k \cdot d$, is often used to compare families with different degrees, to account for the actual cost of allowing additional edges per vertex (or I/O ports per processor). In practical terms, higher degree not only requires more communication lines for a given number of processors but also, for single-chip multiprocessors, decreases the bandwidth available for

each I/O port [14]. To compare infinite families of graphs, we will consider the diameter and product cost as functions of N . In particular, this thesis concentrates on families of small degree, whose diameters grow logarithmically with N . Recent examples of nonlogarithmic families, in which the diameter is N^r for some fixed $r < 1$, include the chordal ring networks [15], the twisted torus [16], and the snowflake [17]. For practical multicomputers, a family of multicomputer graphs will need to have small degree and, ideally, constant degree (so that the interface design does not depend on the number of processors in the network).

Two other important, but less easily quantified, cost criteria are the ease of interpolating graph families and of routing messages within given graphs. The graph families presented here will usually be defined either for all values of N that are a multiple of some constant c , or for all values of N that are a power of some c . We call the former linearly interpolable, the latter exponentially interpolable, and c the interpolation factor. Families that are defined for $N = mM$, where M is multiple of some constant c and m is proportional to $\lg M$, are called semi-linearly interpolable. In other words, the sequence of sizes is arithmetic for linearly interpolable families, geometric for exponen-

tially interpolable families, and has the general form $i \log i$ for semi-linearly interpolable families. To reduce design constraints and simplify expanding existing systems, linear interpolability is preferable to exponential, and small factors to large ones. In part because of the importance of routing, most proposed multi-computer graph families have reasonable routing algorithms, so we will only mention routing occasionally (for example, when proposing new families). Further criteria, relating to reliability, will be discussed in chapter 4.

2.2. Graph Product Families

Several proposed families of graphs can be defined by the Cartesian product of graphs. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, with diameters k_1 and k_2 , degrees d_1 and d_2 , their Cartesian product, $G_1 \times G_2$, is a graph with vertices $V_1 \times V_2$ and edges such that (v_1, v_2) is adjacent to (v_1, w_2) if and only if one of these two conditions holds: $v_1 = w_1$ and $(v_2, w_2) \in E_2$, or $v_2 = w_2$ and $(v_1, w_1) \in E_1$. There are $N_1 N_2$ vertices in the resulting graph, which has diameter $k = k_1 + k_2$ and degree $d = d_1 + d_2$. If we take the Cartesian product of K_2 with itself n times, we have the binary hypercube [18] with $N = 2^n$, $k = n$, and $d = n$. The diameter and degree of the hypercube are both $\lg(N)$, forcing the degree to increase with N .

This product cost, $\lg^2(N)$, can be improved by using different initial graphs in the construction. Given any graph G , we can build an infinite family G_i by defining G_1 to be G and G_i to be $G \times G_{i-1}$. The resulting family has $N_i = N^i$, $k_i = k_1$, and $d_i = d_1$, so that the product cost for G_i is $k d_i^2 = (k d_1^2 / \lg^2(N)) \cdot \lg^2(N_i)$. The vertices of G_i can be assigned addresses that are strings of i base- N digits, allowing a simple routing algorithm: Change each source digit in turn to the destination value by holding all other coordinates fixed, and route

according to the initial graph G to alter the current digit. In the worst case, k steps are required to modify each of the i digits. Since $N_i = N^i$, the family is exponentially interpolable with factor N , so initial graphs with small values of N may be preferable. For $N \leq 10$, the lowest cost is attained using the Petersen graph, which gives $k \cdot d \approx 0.54 \lg^2(N_i)$; for $N \leq 100$, $0.37 \lg^2(N_i)$ can be achieved using a graph with degree 4, diameter 4, and 95 vertices discovered by Bermond [19]. By using sufficiently large initial graphs, the later constructions in this chapter show that product graphs can attain an arbitrarily low coefficient of $\lg^2(N)$; however, this procedure succeeds only because the later families have $O(\lg N)$ costs.

2.2.1. The complete hypercube

The problems of exponential interpolability can be alleviated by allowing several different graphs in the product [20]. For two popular families built in this fashion, we can determine the best possible costs by choosing appropriate component graphs. One form of generalized hypercube [21] uses products of complete graphs. If the component complete graphs all have the same number of vertices, N_1 , the vertices are given N_1 -ary addresses; if the component graphs have various

sizes, say $N_1 \dots N_r$, the vertices are given addresses in a mixed radix. In any case, for each vertex there are edges to all vertices whose addresses differ in exactly one digit. This complete hypercube then has, in the general case, $N = N_1 \cdot \dots \cdot N_r$, $k = r$, and $d = \sum (N_i - 1)$. The mixed radix form provides greater freedom in N , but does not produce denser graphs: For fixed r , elementary calculus shows that the least degree is attained when all the N_i are equal. So for given N , $k \cdot d = r \cdot d \geq r^2 \cdot (N/r - 1)$. Changing the variable to $P = N/r$ shows that $k \cdot d \geq \log_2^2(N)$ ($P-1$) = $(P-1)/\lg^2(P) \lg^2(N)$. The minimum possible product cost for any N is thus achieved by the same P , the one for which $(P-1)/\lg^2(P)$ is minimal. As P must be > 1 , this function has a unique minimum value of approximately 0.741886, when P has a value of approximately 4.922. While this lower bound is unattainable in practice, as all factors must be integers, a product cost of .741929 $\lg^2(N)$ is attained for Cartesian products of K_5 , leaving $N = 5^r$, $k = \log_5(N)$ and $d = 4k$.

2.2.2. The mesh-connected hypercube

The other extreme in terms of component graphs are the mesh-connected hypercubes, in which each component graph of the product is a cycle (or K_2 , as cycles have

at least 3 vertices). The formulae for degree and diameter are then more complex, but the best possible product cost is exactly the same as for the complete hypercubes. Each K_2 component contributes 1 to the degree and to the diameter, while increasing N by a factor of 2. Each cycle C_1 contributes 2 to the degree and $\lfloor 1/2 \rfloor$ to the diameter, and increases N by a factor of 1. Any even cycle could be replaced by an odd cycle, giving the product graph greater N for the same degree and diameter. Similarly, any two K_2 components could be replaced by a single C_3 component for an increase in N without affecting the degree or diameter. Thus, we can find the lower bound for cost as a function of N by using only the formulae appropriate for odd cycles: Given r cycles with $N_1 \dots N_r$ vertices, respectively, the product diameter is $\sum (N_i - 1)/2$ and the degree is $2r$. As before, treating the N_i as continuous shows that the minimum cost is attained when all are equal, giving, once again, a lower bound for the product cost of $2r^2(N_1 - 1)/2 = r^2(N_1/r - 1)$.

2.3. Tree Variations

The densities of the graph product families are surpassed by several families of graphs defined directly in terms of trees. For given degree $d > 2$, the simplest such family has a root vertex with d children, while its other internal vertices have $d-1$ children. If the root is assigned level 0 and the leaves level L , a tree has $d(d-1)^{L-1}$ leaf vertices (each with degree 1), and a total of $(d(d-1)^L - 2)/(d-2)$ vertices. Its diameter is $2L$, so that $k = (2/\lg(d-1)) \lg(N) + O(1)$ and $k \cdot d = (2d/\lg(d-1)) \lg(N) + O(d)$. Routing is extremely simple, as is interpolation: any number N of vertices may be connected within diameter $2 \lceil \lg N / \lg(d-1) \rceil$ by adding a ragged lowest level. In terms of all the criteria used in this chapter, tree families are superior to the graph product families. (Of course, other criteria, such as the reliability issues considered in chapter 4, may make the tree less appealing.)

2.3.1. Multi-tree Structures

The low degree of a tree's leaf vertices can be exploited in several ways to improve density. Friedman [22] suggested joining d copies of the tree at the leaves, producing a regular graph with $N = 2d((d-1)^L - 1)/(d-2)$ and $k = 2L$, which reduces the lower-

order terms in the cost formulae, but does not improve the coefficient of $\lg(N)$. Arden and Lee proposed the family of Multi-Tree-Structures [23], in which some number, m , of component trees are interconnected at the top by a cycle among the roots, forcing each root vertex to have only $d-2$ children, and at the bottom by a cycle among the leaves of all the trees. When d is greater than 3, they also require that each leaf have enough additional connections to other leaves to make the graph regular, but these additional edges are not further specified. Although their exploration of possible Multi-Tree-Structures for degree 3 did uncover several graphs with small diameter for various small values of N , the best general bound they could establish was that, for degree 3, there are Multi-Tree-Structures with $N = 2^{(k+3)/2} + 2^{(k+3)/4}$. Thus, they further improved the low-order terms in the relationship between cost and N , but did not improve the coefficient of $\lg(N)$.

2.3.2. The hypertree family

The most successful tree-based multicomputer graphs, in terms of the cost criteria considered in this chapter, are the Hypertree families of Goodman and Sequin [24]. All the hypertree families are built from complete binary trees with L levels (that is, with the

leaves at level L), with additional hypercube-like edges being added to the vertices to lower the diameter. Hypertree m adds m edges to each vertex, carefully chosen so that the diameter is $\lfloor L(m+2)/(m+1) \rfloor$. The root has degree 2, the other internal vertices have degree $3+m$, and the leaves have degree $1+m$. The vertices in level i are assigned i -bit binary addresses; the additional edges connect vertices that differ in particular bits, depending on m and the level. More explicitly, given $x = x_0 x_1 \dots x_{n-1} \in 2^n$, define $\text{Smallest}(x)$ to be the least i such that $x_{n-i} = 1$. Define $\text{Complement}(x, i)$ to be x with bit i complemented. Then, in particular, Hypertree 1 has degree four and connects vertex v in level i with $\text{Complement}(v, \lfloor 1/2 \text{Smallest}(1) \rfloor)$. Similar, but more complex, definitions can be given when more edges are added.

For a hypertree with parameter m , $k = (m+2)/(m+1) \lg(N) + O(1) = (d-1)/(d-2) \lg(N) + O(1)$, and $k \cdot d = d(d-1)/(d-2) \lg(N) + O(d)$. The least product cost is then achieved by Hypertree 1, where $d = 4$ and $k \cdot d = 6 \lg(N) + O(1)$. This cost is a great improvement over the costs of the graph product families, but is not as good as many other families. Hypertrees also suffer from somewhat more complex routing algorithms,

but, like the other tree families, are linearly interpolable with factor 1; They can be constructed for any value of N by adding a ragged lower edge.

2.4. Single-Exchange Graphs

The results of the graph product families and of the tree variations have been equaled or surpassed by several related families, such as shuffle-exchange graphs, de Bruijn networks, cube-connected-cycles, and the lens. All these families are specific instances of a general class that we call single-exchange graphs. The single-exchange construction is of some interest because it unifies so many popular multicomputer graph families and provides a common cost bound for them. Its inclusion here, however, is justified more by its relationship with the novel double-exchange construction developed below, and by the improved costs that can be achieved by exploiting the construct's generality. The definition provides linearly interpolable versions of several families that were originally proposed in an exponentially interpolable form.

2.4.1. Definitions

To provide a more intuitive understanding of the following definitions, we present the specific case of the standard binary shuffle-exchange multicomputer graph in parallel with the general definitions. The general definitions assume we have been given an arbitrary base $b > 1$ and some number $N > 1$ that is a multiple of b . In

our specific example, b is 2, and $N = 2^n$ for some $n > 0$.

For any $0 \leq x < N$, define $p(x)$ to be $(xb) \bmod N + xb \div N$. $xb \div N$ plays the role of the high-order digit in a b -ary representation of x . In our example, p has the effect of rotating the high-order bit of the binary representation of x to the low-order position, and so becomes the familiar shuffle [25,26] operation. p maps the integers $\{0, N-1\}$ into $\{0, N-1\}$ and, if N is a multiple of b , is a bijection on $\{0, N-1\}$ with inverse $p^{-1}(x) = (x \bmod b) \cdot (N/b) + (x \div b)$.

For any $0 \leq x < N$ and any $0 \leq i < b$, $SE_i(x) = (x \div b) \cdot b + ((x \bmod b) + i) \bmod b$. This single-exchange operation can be viewed as replacing the low-order digit, x_{n-1} , in the b -ary representation of x with the digit $(x_{n-1} + i) \bmod b$. In particular, $SE_0(x) = x$. For our specific example, $SE_1(x)$ complements the least significant bit of x . Because N is a multiple of b , $SE_i(x)$ is a bijection on $\{0, N-1\}$ for any i , with $SE_{-1}(SE_1(x)) = x$.

Using these definitions, we say an undirected graph is a single-exchange graph with S stages, base b , stage-size M , and cost C if its vertices can be assigned unique addresses of the form $\{s, x\}$, where $0 \leq s < S$ and $0 \leq x < M$, such that, $\forall$ vertices $\{s, x\}$ and $\forall 0 \leq i < b$, there is a path of length at most C from $\{s, x\}$ to $\{s+1$

$\bmod S, SE_i(p(x))\}$. (If we view paths in a graph as operations on addresses, C is the cost of rotating in a new digit.) A vertex with address $\{s, x\}$ is said to belong to stage s . Following our general arithmetic conventions, the stage portion of an address is always interpreted $\bmod S$, the string portion is $\bmod M$, and the subscript of any SE operation is $\bmod b$. In the shuffle-exchange example, S is 1, b is 2, M is N , and C is 2.

2.4.2. The diameter of a single-exchange graph

Two more definitions then allow a simple expression for an upper bound on the diameter of any single-exchange graph G with cost C . Let $n = \lceil \log_b N \rceil$, and define $\text{Excess}(s, n)$ to be the maximum over all i such that $0 \leq i < s$ of $\min\{(ai+bn) \bmod b \mid 0 \leq i < s, a = \pm 1, b = \pm 1\}$. Intuitively, $\text{Excess}(s, n)$ is the maximum number of stages that must be traversed in order to reach a stage exactly n stages away from a desired destination. We can now prove the following bound on the diameter of G :

Theorem 1. A single-exchange graph G with cost C has diameter k bounded by $C \cdot (n + \text{Excess}(S, n))$.

Of course, this bound is interesting only because we can construct infinite families of single-exchange

graphs with small, bounded, values of C ; any connected graph could trivially be considered a single-exchange graph with $b = M = N$, $S = 1$, and $C = k$. For the shuffle-exchange, $\text{Excess}(1, n)$ is zero so the diameter bound is $2 \lg N$. Incidentally, this theorem also shows that G is connected.

The proof will follow from four lemmas. Lemma 1.1 shows that, for any j , a vertex $\{s, x\}$ is within distance C of $\{s+1, xb+j\}$; Lemmas 1.2 and 1.3 use this result to show any vertex in stage s is within distance nC of any vertex in stages $s+n$ or $s-n$; and Lemma 1.4 completes the proof by noting that any vertex is within distance $C \cdot \text{Excess}(S, n)$ of some vertex exactly n stages away from any desired destination.

Lemma 1.1. For any vertex $\{s, x\}$ and any j , there is a path of length bounded by C from $\{s, x\}$ to $\{s+1, xb+j\}$.

In terms of our example, this lemma states that a zero or one bit can be shifted in from the right in two or fewer steps.

Proof. Choosing i in the definition of the SE operation such that $SE_i(p(x)) = (xb+j)$ modulo M will yield the desired result. Let $i = j - (xb) \div M$. Then

$$p(x) = (xb) \div M + (xb) \text{ modulo } M$$

and

M

$$\begin{aligned} SE_i(p(x)) &= (xb) \text{ modulo } M + \\ &= ((xb) \div M + j - (xb) \div M) \text{ modulo } b \\ &= (xb+j) \text{ modulo } M. \end{aligned}$$

M

Lemma 1.2. Given any vertex $\{s, x\}$, there is a path to any vertex $\{s+n, y\}$ of length nC , for any $0 \leq y < M$.

For our example, there is only one stage, and this lemma says we may reach any address within $2n$ steps by shifting in the destination address bits from the right.

Proof. Induction on n applications of Lemma 1.1 shows there is a path to any vertex $\{s+n, y\}$ of length bounded by nC , where $y = x \cdot b^n + i_1 \cdot b^{n-1} + i_2 \cdot b^{n-2} \dots + i_n$, modulo M , for some coefficients $i_1 \dots i_n$ such that $0 \leq i_j < b \forall j$. However, the term $x_i b^{n-i}$ is merely an n -digit string of b -ary digits, so it can have any value between 0 and $M-1$, inclusive, allowing y to have any value in that range as well.

M

Lemma 1.3. Given any vertex $\{s, x\}$, there is a path to any vertex $\{s-n, y\}$ of length nC , for any $0 \leq y < M$.

Proof. G is an undirected graph, so we can use Lemma 1.2 starting from $\{s-n, y\}$.

M

Lemma 1.4. Given any vertex $\{s, x\}$ and any stage $0 \leq m < S$, there is a path to some vertex belonging either to stage $m+n$ or to stage $m-n$ of length bounded by $C + \text{Excess}(n, S)$.

S is 1 for our specific example, $\text{Excess}(n, 1)$ is zero, and this lemma merely reflects the fact that no sequence of ρ operations is needed to change the stage.

Proof. By Lemma 1.1, there is a vertex in stage $s+t$ within distance tC of $\{s, x\}$. In particular, there is a vertex in stage $m+n$ within distance $C \cdot \min(s, m+n) \bmod S$, $(m+n)-s \bmod S$ of $\{s, x\}$. Similarly, there is a vertex in stage $m-n$ within distance $C \cdot \min(s, m-n) \bmod S$, $(m-n)-s \bmod S$ of $\{s, x\}$. $\text{Excess}(S, n)$ is defined as the maximum over all $0 \leq i < S$ of $\min((i-n) \bmod S, (i+n) \bmod S, (n-i) \bmod S, (-i-n) \bmod S)$. By considering the case $i = s-m$, we see that $\text{Excess}(S, n)$ is at least the minimum of the distances from $\{s, x\}$ to vertices in stages $m+n$ and $m-n$.

Proof of Theorem 1.

Lemma 1.4 guarantees a path of length bounded by $C \cdot \text{Excess}(S, n)$ from any source vertex to some vertex exactly n stages away from the destination; from here, Lemmas 1.2 and 1.3 guarantee a path to the destination

of length bounded by Cn , completing the proof of Theorem 1.

In several constructions below, we will use the following corollary. M

Corollary 1.5. If there is some constant $C_+ < C$ such that each vertex in stage s is within distance C_+ of some vertex in stage $s+1$, then k is bounded by $Cn + C_+ \cdot \text{Excess}(S, n)$

Proof. A straightforward modification of the proof of Lemma 1.4. M

2.4.3. Routing in single-exchange families

Many single-exchange graph constructions prove the validity of their cost parameter C by providing explicit paths of length $\leq C$ from any vertex $\{s, x\}$ to the vertices $\{s+1, S+1, \rho(x)\} \forall i$. In this case, the diameter proof just given also provides a routing algorithm that is guaranteed to find a path of distance $\leq C + (n + \text{Excess}(S, n))$ between any two vertices.

Algorithm.

Input: A pair of vertices u and v , with addresses $\{s, x\}$ and $\{t, y\}$.

Output: A path from u to v of length $\leq C \cdot (n + \text{Excess}(S, n))$, where $n = \lfloor \log_b M \rfloor$.

Method: Compute $d^+ = \min(s - (t+n)) \bmod S$, $(t+n) - s \bmod S$ and $d^- = \min(s - (t-n)) \bmod S$, $(t-n) - s \bmod S$.

If $d^+ \leq d^-$,

If $(t+n-s) \bmod S \leq (s - (t+n)) \bmod S$,

define $z_0 = \rho^{d^+}(x)$, and construct a path P_0 from u to $\{t+n, z_0\}$ by applying the known routing for $\{s+1, SE_0(\rho(x))\} d^+$ times to u .

Otherwise,

define $z_0 = \rho^{-d^+}(x)$, and construct a path P_0 from u to $\{t+n, z_0\}$ by applying the known routing for $\{s-1, \rho^{-1}(SE_0(x))\} d^+$ times to u .

Compute $y - z_0 b^n \bmod M$, and represent it as a b -ary string $i_0 i_1 \dots i_{n-1}$. For $j = 1$ through n , define $z_j = bz_{j-1} + i_{j-1} \bmod M$. z_n is then equal to $z_0 b^n + \sum_{i=0}^{n-1} i_i b^{n-i} \bmod M = z_0 b^n + (y - z_0 b^n) \bmod M = y$. For $1 \leq j \leq n$, construct a path P_j from z_{j-1} to z_j of length $\leq C$ using the known paths for $SE_h(\rho(z_{j-1}))$, where $h = (i_{j-1} - bz_{j-1} \bmod M) \bmod S$.

b. (See Lemma 1.1.)

If $d^+ > d^-$,

If $(t-n-s) \bmod S \leq (s - (t-n)) \bmod S$,

define $z = \rho^{d^-}(x)$, and construct a path P_0 from u to $\{t-n, z\}$ by applying the known routing for $\{s+1, SE_0(\rho(x))\} d^-$ times to u .

Otherwise,

define $z = \rho^{-d^-}(x)$, and construct a path P_0 from u to $\{t-n, z\}$ by applying the known routing for $\{s-1, \rho^{-1}(SE_0(x))\} d^-$ times to u .

Compute $z - yb^n \bmod M$, and represent it as a b -ary string $i_0 i_1 \dots i_{n-1}$. Set $z_0 = y$ and, for $j = 1$ through n , define $z_j = bz_{j-1} + i_{j-1} \bmod M$. z_n therefore $= yb^n + \sum_{i=0}^{n-1} i_i b^{n-i} \bmod M = z$. For $1 \leq j \leq n$, construct a path P_j from z_{n-j+1} to z_{n-j} of length $\leq C$ by tracing backwards along the known (undirected) paths for $SE_h(\rho(z_{n-j}))$, where $h = (i_{n-j} - bz_{n-j} \bmod M) \bmod S$.

The desired path of length $\leq C \cdot (n + \text{Excess}(S, n))$ is then $P_0 P_1 \dots P_n$.

2.4.4. Specific single-exchange families

2.4.4.1. Cube-connected-cycles

Given integers $m \geq n > 0$, a cube-connected-cycles graph [27] with $m2^n$ vertices has vertex addresses of the

form $\{c, x\}$, where $0 \leq c < m$ and $x \in 2^n$. Each vertex $\{c, x\}$ has edges to $\{c+1 \text{ modulo } m, x\}$, $\{c-1 \text{ modulo } m, x\}$ and, if $c < n$, $\{c, \text{Complement}(x, c)\}$. As before, $\text{Complement}(x, i)$ is x with bit i complemented. Intuitively, a cube-connected-cycles is an n -dimensional binary hypercube, with each vertex replaced by a cycle of n or more vertices. A vertex $\{c, x\}$ has at most three edges, so a cube-connected-cycles is always trivalent. (Preparata and Vuillemin originally defined the cube-connected-cycles only for m a power of two.)

Despite the use of Complement, a cube-connected-cycle with $m = n$ is, in fact, a single-exchange graph. Renumber each vertex $\{c, x\}$ to have address $\{c, p^c(x)\}$. The resulting graph has edges from $\{c, x\}$ to $\{c+1 \text{ modulo } m, p^{c+1}(x)\}$, $\{c-1 \text{ modulo } m, p^{c-1}(x)\}$, and $\{c, SP_1(x)\}$. The associated costs are $C = 2$ and $C_+ = 1$; $\text{Excess}(n, n) = \lfloor n/2 \rfloor$, so, by Corollary 1.5, the diameter is bounded by $\lfloor 5n/2 \rfloor$. Ad hoc arguments show that the diameter is actually 1 for $n = 1, 4$ for $n = 2, 6$ for $n = 3$, and $\lfloor 5n/2 \rfloor - 2$ for $n > 3$. Thus, $k = 5/2 \lg(N) + O(\lg \lg(N))$, and $k \cdot d = 15/2 \lg(N) + O(\lg \lg(N))$.

Preparata's original definition appears to be exponentially interpolable; however, the diameter bound and routing characteristics are preserved when our gen-

eral single-exchange definitions are used, producing a semi-linearly interpolable family with factor 2. Cube-connected-cycles with $m > n$ can also be mapped into a version of the single-exchange construction, by using the same renumbering for $c < n$ and leaving the addresses of the vertices with $c \geq n$ unchanged. Such graphs are less dense, so this variation of the multistage single-exchange construction has not been pursued.

2.4.4.2. The lens

The lens family was originally proposed for a multicomputer model with busses instead of point-to-point communications [28]. However, as Finkel and Solomon point out, a completed lens with b busses per processor and b processors per bus can also be viewed as a regular graph of degree $2b$. Given parameters b and $n > 1$, a lens in this interpretation can be defined that has nbn vertices, degree $2b$, and diameter $= \lfloor 3n/2 \rfloor$ as follows: Given $x = x_0 x_1 \dots x_{n-1} \in b^n$, define $\text{NewBit}(x, m, i) = x_0 x_1 \dots x_{m-1} i x_{m+1} \dots x_{n-1}$, that is, x with the m -th bit replaced with i . Assign the vertices unique addresses of the form $\{s, x\}$, where $0 \leq s < n$, and $x \in b^n$. Vertex $\{s, x\}$ is then connected to vertex $\{s+1 \text{ modulo } n, \text{NewBit}(x, s, i)\}$ $\forall 0 \leq i < b$, and to vertices $\{s-1 \text{ modulo } n, \text{NewBit}(x, s-1, i)\}$ $\forall 0 \leq i < b$. This con-

struction has been deliberately written to show its similarity with the cube-connected-cycles; the same automorphism taking $\{a, x\}$ to $\{s, p^a(x)\}$ produces a multistage single-exchange graph with $s = n$, base b , $d = 2b$, and $C = 1$. By Theorem 1, the diameter is bounded by $n + \text{Excess}(n, n) = \lfloor 3n/2 \rfloor$; ad hoc arguments again show that this bound is exact. Thus, $k = 3/(2 \lg(d/2)) \lg(N) + O(1)$ and $k \cdot d = 3d/(2 \lg(d/2)) \lg(N) + O(d)$.

When cast into the form of our single-exchange definitions, this family is semi-linearly interpolable with interpolation factor b , thereby answering the question of partial lenses proposed by Finkel and Solomon [28]. The same family of graphs, in the exponentially interpolable version, has been proposed for multicomputers by Farhi, under the name C_S graphs [29]. By designating most stages switches instead of processors, the base-2 lens can be interpreted as the popular permutation network variously called [30,31] the "multistage shuffle-exchange" [32,26], the S - W banyan with $s = p = 2$ [33], the omega [1], and the indirect binary n -cube [34].

2.4.4.3. The shuffle-exchange graph

The family of multicomputer graphs called shuffle-exchange graphs [2,3,4] has already appeared in sections

2.4.1 and 2.4.2 as a specific example of the single-exchange definition. The family is defined for $N = 2^n$ by assigning unique addresses in 2^n to the vertices, and connecting vertex v to $p(v)$, $p^{-1}(v)$, and $SE_1(v)$. The resulting graphs are clearly trivalent single-exchange graphs with $s = 1$ and $C = 2$. Theorem 1 shows the diameter bounded by $2n = 2 \lg(N)$; ad hoc arguments show the actual diameter is $2 \lg(N) - 1$, for a product cost of $6 \lg(N) - 3$.

2.4.4.4. The de Bruijn networks

For all even degrees above 3, the densest previously-known infinite families are the undirected versions of de Bruijn networks [35], where $k = 1/\lg(d/2) \lg(N)$ and $k \cdot d = d/\lg(d/2) \lg(N)$. Although de Bruijn networks must have even degree, their asymptotic costs were until recently not only less than the costs of any previously published family of the same degree d , but also less than those of any family with degree $d + 1$. The double-exchange graphs introduced later in this chapter now excel for degrees 4 and 5, while a variation of de Bruijn graphs proposed by Farhi [29], called C_S^* graphs, now excels at odd degrees of 7 or higher. For other recent variations and rediscoveries of de Bruijn networks, see Stone [36], Golunkov [37], Lam [38], Imase

and Itoh [39], and Pradhan [40].

Given a base $b > 1$ and some number N of vertices, where N is a multiple of b , a de Bruijn network with degree $2b$ is constructed by assigning each vertex a b -ary address and connecting vertex x with vertices $SE_1(p(x))$ $\forall 0 \leq i < b$. De Bruijn's original construction was a directed graph with N a power of the base b ; when the construction is used for undirected graphs, some b^2 vertices (independent of N) have degree less than $2b$, due to the specification of self-loops and multiple edges involving the vertices $(az)^b$ or $(az)^a$ for arbitrary b -ary digits a and z . The edges of the original directed de Bruijn network provide exactly the same single-step processor-to-processor interconnections as the permutation network sometimes called the single-stage shuffle-exchange [26,36].

The undirected de Bruijn network is a single-exchange graph with $S = 1$ and cost parameter $C = 1$. Theorem 1 shows that the diameter is bounded by $\lceil \log_b N \rceil$; the observation that changing one b -ary digit requires at least 1 edge proves that the diameter is at least $\lceil \log_b N \rceil$. The diameter k is therefore $1/\lg(b) \lg(N) = 1/\lg(d/2) \lg(N)$, and $k \cdot d$ is $d/\lg(d/2) \lg(N)$. Interestingly, the ratio of either cost measure for the de Bruijn to the optimum possible is $\lg(d-1)/\lg(d/2)$,

which approaches 1 as d approaches infinity. Until the development of our double-exchange graphs, the degree-6 de Bruijn had the least product cost ($3.785 \lg N$) of any family known.

The corresponding multistage single-exchange graph corresponds closely to the permutation network called the multistage shuffle-exchange. A de Bruijn with degree $2b$, S stages, and stage-size M has, by Theorem 1, diameter bounded by $n + \text{Excess}(S, n)$, where $n = \lceil \log_b M \rceil$. When $S = n$, the family includes the lens; when M is a power of b , Farhi [29] calls these C_S graphs, and proves that the diameter is in fact equal to $n + \text{Excess}(S, n)$.

2.4.4.5. The C'_S graph

For odd degree at least 5, Farhi proposes the following variation, the C'_S graph, which gives $k = 2/\lg(\lfloor d/2 \rfloor \cdot \lceil d/2 \rceil) \lg(N) + O(1)$. Let $b = \lfloor d/2 \rfloor \cdot \lceil d/2 \rceil$, and let b_1 and b_2 denote $\lfloor d/2 \rfloor$ and $\lceil d/2 \rceil$ respectively. The graph consists of $Sb_1^{b_1}b_2^{b_2}$ vertices with addresses of the form $\{s, r, y\}$, where s is the stage, x is a string of n b_1 -ary digits, and y is a string of n b_2 -ary digits. When s is even, $\{s, x, y\}$ has edges to $\{s+1, SE_1(p(x)), y\}$ and $\{s-1, x, p^{-1}(SE_2(y))\}$, where $0 \leq i < b_1$ and $0 \leq j < b_2$; when s is odd, $\{s, x, y\}$ has edges to $\{s+1, x, SE_2(p(y))\}$ and $\{s-1, p^{-1}(SE_1(x)), y\}$. Any vertex can

reach any other in a stage $2n$ stages away by alternately moving in the x and y digits of the destination; other vertices are reached by first routing to any vertex whose stage is $2n$ away from the destination.

At degree 5, the C'_S family has diameter $k = (2/\lg 6) \lg N \approx 0.774 \lg N$ and $k \cdot d \approx 3.869 \lg N$. Until the development of our double-exchange graphs, this family had the least diameter of any degree-5 family known. Unlike the lens, a single-stage version of the C_S graphs is not possible: S must be even. Fathi's original definition requires M to be a power of 2, but our general p operation (instead of the original rotate) makes the families linearly interpolable with factor $2b$ -- M can be any multiple of b , while S is at least 2.

2.4.5. Operations that construct single-exchange families

We can define several natural operations on graphs that build infinite families of single-exchange graphs whose costs and routing algorithms are determined directly from the initial graph. The simplest operation on a given graph G with N_G vertices, degree d_G and diameter k_G is to build a single-exchange graph with S stages and any multiple, say m , of N_G vertices per stage. Arbitrarily assign unique addresses from the set

$\{0, 1, \dots, N_G - 1\}$ to the vertices of G . Let the base $b = N_G$, the stage-size $M = mb$, and $n = \lceil \lg M \rceil$. The vertices in the single-exchange graph are the formal pairs $\{s, x\}$, where $0 \leq s < S$ and $0 \leq x < M$. Connect vertex $\{s, x\}$ to vertices $\{s+1, p(x)\}$, $\{s-1, p^{-1}(x)\}$, and $\{(s, y) \mid x \text{ modulo } b \text{ is adjacent to } y \text{ modulo } b \text{ in } G\}$. The resulting single-exchange graphs have degree $d = 2 + d_G$ and mbs vertices. The value of x modulo b (intuitively, the least significant digit of x) can be changed by using the edges of G ; a p requires a single edge, so $C = 1 + k_G$, $C_+ = 1$, and, by Corollary 1.5, $k \leq (1 + k_G)n + \text{Excess}(S, n)$. This construction immediately creates the shuffle-exchange family by taking $G = K_2$ and $S = 1$, and the cube-connected-cycles by taking $G = K_2$, a stage-size of 2^n , and n stages. It also produces a degree 5 family with $k \leq 3/\lg(10) \lg(N) \approx .903 \lg(N)$, by using the Petersen graph for G ; the only degree-5 families known with lower diameter are the C'_G family and our double-exchange family.

A slight generalization includes all de Bruijn networks and all lenses. Given G , choose an integer $e > 0$ and construct a single-exchange family with degree $d + 2e$, base $b = eN_G$, S stages, and stage-size mb for any m by defining the vertices as pairs $\{s, x\}$ as before. Define $\text{CopyNum}(x) = (x \text{ modulo } b) \text{ div } N_G$ and $\text{Element}(x) =$

(x modulo b) modulo N_G . Each digit of x may be thought of as encoding e copies of G ; in effect, $\text{CopyNum}(x)$ selects one copy of G from the least significant digit of x , and $\text{Element}(x)$ determines a particular vertex in G . Connect vertex $\{s, x\}$ with vertices $\{s+1, y\} \mid \text{Element}(y) = \text{Element}(\rho(x)) \mid \{s-1, y\} \mid \text{Element}(y) = \text{Element}(\rho^{-1}(x))\}$ and $\{s, y\} \mid \text{CopyNum}(y) = \text{CopyNum}(x)$ and $\text{Element}(y)$ is adjacent to $\text{Element}(x)$ in G . The cost parameters C and C_+ of this single-exchange family are again $l+k_G$ and l , respectively, so by Corollary 1.5, the diameter is bounded by $(l+k_G)n + \text{Excess}(S, n)$. By picking $G = K_1$, and $S = 1$, we create the degree $2e$ de Bruijn networks; choosing $S = n$ produces the corresponding lens family. The diameter bound provided by Corollary 1.5 is tight for the cube-connected-cycles, the shuffle-exchange, the de Bruijn, and the lens; as mentioned above, ad hoc arguments show that the actual diameters are at most 2 less than our general bound for single-exchange families.

The graph operations just defined, together with Theorem 1 and its corollaries, allow any ad hoc inexpensive graph to be used to generate a linearly interpolable infinite family of inexpensive graphs. The constructions provide a simple routing algorithm for each family generated, for which one only needs to know how

to route in the original graph G : The destination address is shifted in, digit by digit, using the routing in G to implement the SE_i operations and our general ρ operation for the rotations. However, despite the recent progress in constructing ad hoc dense graphs for small diameters [41,42,29,43,44,45,46,19], these single-exchange graph operations cannot yet surpass the de Bruijn, C'_G , and double-exchange families.

2.5. Double-Exchange Graphs

Surprisingly, a slight change in the single-exchange construction produces quite different -- and often superior -- graphs. We will first define this construction for single-stage double-exchange graphs in which N is a power of the base, $N = b^n$. The requirement that N is a power of the base not only simplifies the definitions, but also allows us to prove a lower diameter bound. Our construction uses the same rotate operation, ρ , but introduces a new "exchange-like" operation that we call the double-exchange.

2.5.1. Definitions

Given $x = x_0 x_1 \dots x_{n-1} \in b^n$, the digit sum of x , $\text{Sum}(x)$, is $\sum x_i$. (As usual, all arithmetic involving digits of x is modulo b , while arithmetic involving subscripts of its digits is modulo n .) We define $\rho(x) = x_1 x_2 \dots x_{n-1} x_0$ and $\rho^{-1}(x) = x_{n-1} x_0 x_1 \dots x_{n-2}$. If $n \geq 2$, then for all $0 \leq i < b$, the double-exchanges of x are $\text{DE}_i(x) = x_0 x_1 \dots x_{n-3} x_{n-2}^{-1} x_{n-1} + i$. Given an integer $0 \leq i < b$, $x \in b^n$ will denote $x_0 + i x_1 + i \dots x_{n-1} + i$. (So $\text{DE}_0(x) = x = x \oplus 0$.)

Let X be a nonempty set of vertices in a connected graph G . X is a double-exchange set $X_{b,n}$ with costs C_i if there exists a labeling of the vertices of X with

distinct addresses in b^n such that:

- If x and y are both in X , $\text{Sum}(x) = \text{Sum}(y)$.
- $\forall x \in X$ and $\forall 0 \leq i < b$, $\text{DE}_i(\rho(x)) \in X$;
- There are constants C_i such that $\forall x \in X$ and $\forall 0 \leq i < b$, the distance from x to $\text{DE}_i(\rho(x))$ is $\leq C_i$.

If the double-exchange set X includes all vertices in G , G is a double-exchange graph, $G_{b,n}$ with costs C_i . We will show below that any double-exchange set $X_{b,n}$ has b^{n-1} vertices.

Given a vertex v in a double-exchange set, consider a path from v that begins with a ρ and consists of alternating ρ and DE operations: $\rho \text{DE}_{j_0} \rho \text{DE}_{j_1} \dots \rho \text{DE}_{j_k}$. Each j_i satisfies $0 \leq j_i < b$, so the j 's form a b -ary string $j_0 j_1 \dots j_k$. Conversely, any b -ary string x can represent a sequence of DE subscripts. We write x for the corresponding path and $v \circ x$ for the vertex reached from v by following x .

2.5.2. The diameter of a double-exchange set

In this section, we will prove the following:

Theorem 2. A double-exchange set $X_{b,n}$ has b^{n-1} vertices and diameter bounded by $\lfloor \frac{n}{5} \sum C_i \rfloor$.

Let u be a vertex in a double-exchange set $X_{b,n}$. Let v be a string in b^n . Lemmas 2.1 and 2.2 will show

that addresses with equal digit sums are connected by paths of the form $(p \text{ DE})^*$, so that x has b^{n-1} members; Lemma 2.4 will use the paths of Lemma 2.2 to prove the diameter bound.

Lemma 2.1. Given v and $x \in b^n$ such that $v = u \circ x$, $v \neq 0 \leq i < n$, $v_i = u_i + x_i - x_{i+1}$.

Proof. This lemma follows from a tedious but straightforward induction.

□

Lemma 2.2. If $\text{Sum}(u) = \text{Sum}(v)$, there is a path from u to v .

Proof. We will show there exists a string $x \in b^n$ such that $v = u \circ x$; property c will then guarantee that the path x exists. For fixed u and v , Lemma 2.1 provides n linear equations in the n unknowns x_i :

$$x_{i+1} = u_i - v_i + x_i, \quad 0 \leq i < n$$

However, these equations are not independent. Solving in terms of x_0 yields

$$x_i = x_0 + \sum_{j < i} (u_j - v_j), \quad 1 \leq i < n$$

$$\text{and } x_0 = x_0 + \sum_{j < n} (u_j - v_j) + x_0 = \text{Sum}(u) - \text{Sum}(v) + x_0$$

Therefore, $\text{Sum}(u)$ must equal $\text{Sum}(v)$. When this require-

ment is met, setting $x_0 = 0$ determines the remaining x_i such that $v = u \circ x$. For later reference, we note that x_0 may be assigned any value, yielding b distinct solutions of the form $x \oplus j$, $0 \leq j < b$.

□

Corollary 2.3. A double-exchange set $X_{b,n}$ has b^{n-1} members.

Proof. A double-exchange set is defined to be nonempty. Given $u \in X_{b,n}$, Lemma 2.2 shows there is a path from u to an address v using subpaths of the form $\text{DE}_i(p(x))$ whenever $\text{Sum}(u) = \text{Sum}(v)$. Property b then implies that any such v is in X . There are b^{n-1} distinct strings v such that $\text{Sum}(v) = \text{Sum}(u)$, since the first $n-1$ digits may be chosen freely, and the remaining digit chosen to produce the correct digit sum.

□

In particular, Corollary 2.3 shows that a double-exchange graph has $N = b^{n-1}$.

Lemma 2.4. If $\text{Sum}(u) = \text{Sum}(v)$, then there exists a path from u to v of length less than or equal to $\lfloor \frac{n}{b} \rfloor \sum c_i$.

Proof. Let $x \in b^n$ be the string defined in Lemma 2.2, where $v \neq 0 \leq i < b$, $v = u \circ (x \oplus i)$. In order to calculate the minimum distance from u to v along these paths, we

must find the minimum sum of the C_j 's over the b alternative paths. For $0 \leq i < b$, let n_i be the number of j 's such that $x_j = i$. The contribution of the C_i 's to the length of x is $\sum C_j n_{j+i}$.

Suppose the minimum of this sum is greater than $\sum_{j=0}^b \sum C_j$. Then $\forall i, \sum C_j n_{j+i} > \sum_{j=0}^b \sum C_j$. So $\sum (\sum C_j n_{j+i}) > n \sum C_j$. Since all the subscript arithmetic is modulo b , each C_j occurs in a product exactly once with each n_i . We therefore can interchange the summations to show that $\sum C_j (\sum n_i) > n \sum C_j$. But $\sum n_i = n$, because each digit in x is counted exactly once, yielding the contradiction that $\sum C_j n > n \sum C_j$. Therefore there is some i for which the length of x is $\leq \sum_{j=0}^b \sum C_j$. All distances in a graph are integral, so $\text{Distance}(u, v) \leq \left\lfloor \sum_{j=0}^b \sum C_j \right\rfloor$.

M

Proof of Theorem 2 Since all vertices in a double-exchange set have the same digit sum, Lemma 2.4 establishes the diameter bound of Theorem 2.

M

Some later constructions will use the following corollary.

Corollary 2.5. If a graph G contains a double-exchange set $X_{b,n}$ such that any vertex in G is at most distance r from some member of the double-exchange set X , then the

diameter of G is less than or equal to $2r + \text{the diameter of } X$.

Proof. Immediate.

M

Many of the later constructions in fact have a slightly lower diameter bound, although not enough lower to alter the asymptotic costs.

Corollary 2.6. If a double-exchange set $X_{b,n}$ has the property that there is a constant $C_R > 0$ such that $\forall$ vertices x and $\forall 0 \leq i < b$, the distance from x to $DE(x, i)$ is bounded by $C_i - C_R$, then the diameter of X is less than or equal to $\left\lfloor \sum_{i=0}^b \sum C_i \right\rfloor - C_R$.

Proof. We can omit the initial p in any path x , and repeat the proof of Theorem 2. The counterpart of Lemma 2.1 is $\forall 0 \leq i < n, v_i = u_{i-1} + x_i - x_{i+1}$. The counterpart of Lemma 2.2 then yields the n equations

$$x_{i+1} = u_{i-1} - v_i + x_i, \quad 0 \leq i < n$$

These equations are also dependent, and the rest of the proof continues unaltered, producing a final cost that is lower by the use of DE instead of $p \cdot DE$ at the start of the path.

M

2.5.3. Routing in double-exchange families

The proof of Theorem 2 implies a routing algorithm for double-exchange sets that is analogous to the routing algorithm for single-exchange graphs. Because we have restricted our attention to addresses in b^n , however, the algorithm is much simpler.

Algorithm.

Input: A pair of vertices u and v in a double-exchange set $X_{b,n}^n$.

Output: A path from u to v of length $\leq \left\lceil \frac{n}{2} \right\rceil$.

Method: For $y \in b^n$, define

$$\text{Cost}(y) = \sum y_i.$$

Define $x_0 = 0$ and for $i = 0, \dots, n-2$

$$x_{i+1} = u_i - v_i + x_i$$

For $0 \leq j < b$, compute $\text{Cost}(x \oplus j)$. Let $j_{\min}$ be the value of j such that $\text{Cost}(x \oplus j)$ is minimal. This calculation requires no more than $O(bn)$ steps; for any particular family, b is fixed. The desired path is then $x \oplus j_{\min}$.

2.5.4. Specific double-exchange families

The double-exchange construction will allow us to surpass any previously published asymptotic density for degrees 3, 4, and 5. For any b and n , we can construct a double-exchange graph $G_{b,n}$ by providing an edge from each vertex for each of the operations ρ , ρ^{-1} , and DE_i for all $0 < i < b$. Since $d = 1 + b$, Corollary 2.6 ensures that $N = (d-1)^n$ and the diameter is $< (2d-3)/((d-1) \lg(d-1)) \lg(N)$. Fixing $b = 2$ yields a family of degree 3 graphs with $k < 3/2 \lg(N)$, compared to $2 \lg N$ for the best previous trivalent families. For higher degrees, this basic construction does not surpass the de Bruijn density. At degree 4, $k \leq 5/(3 \lg 3) \lg(N) = 1.052 \lg(N)$, in contrast to $k = \lg(N)$ for the de Bruijn network. The comparison worsens as d increases: For large d , $k = 2/\lg d \lg(N)$, compared to $1/\lg \lfloor d/2 \rfloor \lg(N)$ for de Bruijn networks. However, there are construction techniques that produce cheaper infinite families by using paths instead of single edges to represent the operations.

2.5.4.1. Degree 3

The trivalent graphs just constructed with $b = 2$ have $N = 2^{n-1}$ vertices and a diameter bounded by $\lfloor 3n/2 \rfloor - 1$; ad hoc arguments show that the diameter is at

least $\lfloor 3n/2 \rfloor - 2$. In view of the simplicity of their construction and the fact that all previously published infinite trivalent families have diameter at least $2 \lg(N)$, we will study them in more detail in later chapters. The construction defines two graphs for a given n -- one whose addresses have an even number of ones, and one whose addresses have an odd number. When n is odd, each address in one graph has its complement in the other, and the two graphs are isomorphic. When n is even, the graphs are not isomorphic, with the even parity graph having two vertices of degree 1 while the odd parity graph has none. If we always select the even parity graph for each n , the resulting even double-exchange family has a close relationship with the shuffle-exchanges, which is explored in the next chapter.

Our cheapest trivalent variation gives $k/\lg(N) < 7/(3 \lg 3) = 1.472$. For it, we use ternary addresses and two sets of vertices, $V \subseteq 3^n$ and $W \subseteq 3^{n-1}$. A vertex $v \in V$ has two neighbors in V , $\rho(v)$ and $\rho^{-1}(v)$, and one neighbor in W : $v_0 v_1 \dots v_{n-3} v_{n-2} v_{n-1}$. A vertex $w \in W$ has three neighbors, all in V : v , $DE_1(v)$, and $DE_2(v)$, where $v = w_0 w_1 \dots w_{n-3} w_{n-2} 0$. V is a double-exchange set with $C_0 = 1$ and $C_1 = C_2 = 3$, so it has 3^{n-1} members, all within distance $\frac{7}{3}n - 1$. There are 3^{n-2} elements of

W adjacent to these v 's, any two of which lie within distance $1 + \frac{7}{3}n$ of each other, by Corollary 2.5, giving the final graph a diameter bounded $1 + \frac{7}{3}n$ for $4 \cdot 3^{n-2}$ vertices. Its costs are then $k < 1.472 \lg(N)$ and $k \cdot d < 4.417 \lg(N)$.

2.5.4.2. Degree 4

The densest infinite family previously known with degree 4 is the binary de Bruijn network, which has $k = \lg(N)$. We can construct two double-exchange families that have lower cost. By using base 7, graphs can be constructed with $k \leq .967 \lg(N)$: connect vertex v with $\rho(v)$, $\rho^{-1}(v)$, $DE_1(v)$, and $DE_6(v)$. Then $C_0 = 1$, $C_1 = C_6 = 2$, $C_2 = C_5 = 3$, and $C_3 = C_4 = 4$. By Theorem 2, $N = 7^{n-1}$, while by Corollary 2.6 (with $C_R = 1$), $k < \lfloor 19 n/7 \rfloor \leq 19/(7 \lg 7) \lg(N) = .967 \lg(N)$.

An even better infinite family of degree-4 double-exchange graphs is produced using base 5 strings and a double-exchange of plus or minus 1. More precisely, we define a graph in which a vertex whose address is the base 5 string v has neighbors $\rho(v)$, $\rho^{-1}(v)$, $DE_1(v)$, and $DE_4(v)$. This construction makes $C_0 = 1$ and $C_1 = C_4 = 2$. $DE_2 = DE_1 DE_1$ and $DE_3 = DE_4 DE_4$, so $C_2 = C_3 = 3$. Corollary 2.3 shows that $N = 5^{n-1}$, and Corollary 2.6, with $C_R = 1$, requires that $k < \lfloor 11 n/5 \rfloor$, so $k < 11/(5 \lg 5)$

$\lg(N) = .947 \lg(N)$.

2.5.4.3. Degree 5

There are many infinite double-exchange families with degree 5 and $k < \lg(N)$. The best has $k \leq 3/4 \lg(N)$, and is constructed by letting $b = 4$ and connecting vertex v to $p(v)$, $DE_1(p(v))$, and $DE_2(v)$. Since DE_2 is its own inverse, each vertex has five or fewer neighbors. Clearly, $C_0 = C_1 = 1$ and $C_2 = 2$, while $C_3 = 2$, since $p DE_3$ can be implemented by $p DE_1 DE_2$. Lemmas 2.2 and 2.4 then allow 4^{n-1} vertices to lie within diameter $n + \lceil n/2 \rceil$, giving $k \leq 3/(2 \lg 4) \lg(N) = 3/4 \lg(N)$. The resulting $k \cdot d$ of $15/4 \lg(N) = 3.75 \lg(N)$ is the lowest product cost for any published infinite family at any degree.

2.5.5. Variations of the double-exchange

We considered several variations of the double-exchange definition in the hope of improving the diameter bounds still further. Repeating the arguments of lemmas 2.1 and 2.2 shows that increasing the number of digits affected by the exchange to make "triple" or "quadruple" exchange families does not improve the diameter. On the chance that using noncontiguous digits in the exchange would give better diameters, we wrote a

program that explored all possible patterns of exchange within the last 6 bits of the even double-exchange, for n between 6 and 11 inclusive (so N was generally between 32 and 2048, depending on the number of connected components produced). None of these variations had greater density than the double-exchange; the ones that equaled it turned out to be logically equivalent.

There are, however, two related constructions that can produce denser graphs, although without improving the asymptotic value of $k/\lg(N)$. These are the twisted double-exchange and the multistage double-exchange. The diameter bounds for both constructions rely on Theorem 2.

2.5.5.1. The twisted double-exchange

For $x = x_0 x_1 \dots x_{n-1} \in b^n$, define $\text{TwistedRotate}(x) = x_1 x_2 \dots x_{n-1} x_0^{+1}$. A twisted double-exchange set $X_{b,n}$ with costs C_i is defined in exact analogy with a double-exchange set, except that TwistedRotate is used in place of p . Let $C_{\min}$ be the least C_i .

Corollary 2.7. A twisted double-exchange set $X_{b,n}$ has b^n vertices and diameter less than or equal to $b C_{\min} + \lfloor \frac{n}{b} \sum C_i \rfloor$.

Proof. Because each TwistedRotate increments the digit-sum, $\text{Sum}(x)$, by one, a straightforward analogy of the proof of the double-exchange theorem allows any vertex u to reach any address $v \in b^n$ such that $\text{Sum}(v) - \text{Sum}(u) = n$ modulo b , in distance bounded by $\lfloor \frac{n}{b} \rfloor \cdot \sum C_i$. Therefore, any vertex u in X can reach any address v in b^n by an initial sequence of at most b TwistedRotates (a path of length at most $bC_{\min}$), followed by a path of length $\lfloor \frac{n}{b} \rfloor \sum C_i$.

2.5.5.2. The Moebius graph

For small values of the C_i 's, this construction then allows somewhat denser graphs than the untwisted version. In particular, the construction for the degree 3 even-parity double-exchange family given above is improved by using a twisted rotate, producing a family of trivalent graphs with $N = 2^n$ and $k \leq \lfloor 3n/2 \rfloor$. We call the resulting graphs Moebius graphs [47], because of their twist, and study them in more detail in chapter 4.

2.5.5.3. The multistage double-exchange

Multistage versions of double-exchange sets can be defined in exact analogy with the multistage single-exchange graph definition. Given a number of stages S ,

a base b , and a parameter n , we assign addresses of the form $\{s, x\}$ to vertices in the set, where $0 \leq s < S$ and $x \in b^n$. $\text{StageRotate}(\{s, x\})$ is thus $\{s+1, \rho(x)\}$, and $\text{StagedE}_i(\{s, x\})$ is $\{s, \text{DE}_i(x)\}$. A multistage double-exchange set with costs C_i is then defined by replacing the operations in the double-exchange definition with the corresponding staged versions. Again, let $C_{\min}$ be the least C_i .

Corollary 2.8. A multistage double-exchange set with base b , parameter n , S stages, and costs C_i has b^{n-1} vertices and diameter less than or equal to $\lfloor \frac{n}{b} \rfloor \sum C_i + C_{\min} \text{Excess}(S, n)$.

Proof. Using the proof method of Theorem 2, we show that a vertex $\{s, x\}$ can route to any vertex $\{s+n, y\}$ or $\{s-n, y\}$ in distance less than or equal to $\lfloor \frac{n}{b} \rfloor \sum C_i$. Then, as in Theorem 1, we show that we can route to some vertex n stages away from any given destination within distance $C_{\min} \text{Excess}(S, n)$.

□

Corollaries 2.7 and 2.8 improve only the low order terms in the cost functions; however, such a reduction can make a large difference in the number of vertices that a graph of given degree and diameter can have. At degree 3, for example, the most vertices for a given di-

iameter under the various choices are:

```
diameter:      3  4  5  6  7  8  9  10
-----
shuffle-exchange:  4  --  8  --  16  --  32  --
double-exchange:  --  8  16  --  32  64  --  128
twisted DE:       8  16  --  32  64  --  128  256
multistage DE:    10  14  24  72  88  208  256  448
```

2.5.6. Interpolating double-exchange families

The double-exchange constructions presented so far produce families that are highly dense but exponentially interpolable (for a fixed number of stages). If linear interpolability is more critical than diameter, the equivalent linearly interpolable constructions can be defined in analogy with the single-exchange construction. The generalized rotate operation is the same for both constructions, but the definition of the double-exchange is more complex than the single-exchange: Given a base $b > 1$ and some number $N > 1$ that is a multiple of b^2 , define $p(x)$ to be (xb) modulo $N + xb \operatorname{div} N$, and define $DE_i(x)$ to be

$$\begin{aligned} & (x \operatorname{div} b^2) \cdot b^2 + \\ & ((x \operatorname{modulo} b^2) \operatorname{div} b - i) \operatorname{modulo} b) \cdot b + \\ & (x \operatorname{modulo} b) + i) \operatorname{modulo} b. \end{aligned}$$

Intuitively, the first term in the DE expression sets

the two low-order digits of x to 0, the second term subtracts i from the next-to-last digit, and the third term adds i to the last digit. N is a multiple of b^2 , so $DE_i(x)$ is a bijection on $[0, N-1]$ for any i , with $DE_{-1}(DE_i(x)) = x$. Unfortunately, we have not found a tight diameter bound for such constructions. Letting C be the maximum of the costs C_i , we can prove the following:

Corollary 2.9. An interpolated double-exchange set X has diameter less than or equal to $C \lceil \log_b N \rceil$.

Proof. From any vertex u , there is a path of length at most C to $(ub+i) \operatorname{modulo} N$; as in Lemma 1.2, $\lceil \log_b N \rceil$ iterations reach any v .

For the degrees we have explored, the linearly interpolated double-exchange graphs are generally denser than the corresponding single-exchange graphs, and twisted graphs are denser than the untwisted versions. Considering the 250 trivalent graphs with N a multiple of four between 4 and 1000, for example, the mean value of $k/\lg N$ is 1.55 for the shuffle-exchange, 1.49 for the even double-exchange, and 1.37 for the Moebius. The definition can be extended to include multiple stages for further improvements in density; for comparison, we re-

peat the example from the last section with the addition of linear interpolation:

diameter:	3	4	5	6	7	8	9	10
shuffle-exchange:	4	-	8	12	20	68	84	156
double-exchange:	-	8	16	24	36	68	120	188
twisted DE:	8	16	--	36	64	108	180	276
multistage DE:	10	14	24	72	88	208	300	484
multistage twist DE:	16	24	32	80	108	192	260	600

The values for the multistage versions represent the best results found among graphs with not more than 17 stages and not more than 256 vertices per stage. Although it may not represent the densest possible multistage Moebius for diameter 10, the value of 600 vertices represents the densest trivalent graph currently known at that diameter [46].

2.6. Summary

The degrees and costs of the many families of graphs proposed for multicomputer interconnection models are summarized in Table 2.1. The densest have fixed degree and diameters logarithmic in N , the number of processors. Our double-exchange families have notably high density as well as simple routing algorithms. In particular, for degrees three, four, and five, our double-exchange families have the lowest diameters; our best degree five family also has the lowest product cost of any family yet reported.

Chapter 3
Algorithms and Layout

In the previous chapter, we defined the shuffle-exchange and even double-exchange families of trivalent graphs. These families have very similar definitions; the even double-exchange family uses the double-exchange, and the shuffle-exchange family uses the single-exchange. Nevertheless, the even double-exchange has much better diameter: $1.5 \lg N$ instead of $2 \lg N$. This chapter briefly explores the consequences of a homomorphism between the two families that allows the even double-exchange family to exploit known results on exchange graphs.

Throughout this chapter, all strings are in 2^n ; continuing the conventions of Chapter 2, all arithmetic involving bits is modulo 2 and all arithmetic involving subscripts is modulo n . The bitwise complement of v is denoted by $\bar{v}$ or $\sim v$.

Table 2.1
Degrees and Costs of Graph Families

	Diameter	Product Cost
<u>Degree 3</u>		
tree	2 $\lg N$	6 $\lg N$
cube-connected cycles	2.5 $\lg N$	7.5 $\lg N$
shuffle-exchange	2 $\lg N$	6 $\lg N$
double-exchange	1.5 $\lg N$	4.5 $\lg N$
ternary double-exchange	1.472 $\lg N$	4.417 $\lg N$
<u>Degree 4</u>		
tree	1.262 $\lg N$	5.047 $\lg N$
Hypertree 1	1.5 $\lg N$	6 $\lg N$
binary lens	1.5 $\lg N$	6 $\lg N$
binary De Bruijn	1 $\lg N$	4 $\lg N$
double-exchange	0.947 $\lg N$	3.790 $\lg N$
<u>Degree 5</u>		
tree	1 $\lg N$	5 $\lg N$
Hypertree 2	1.333 $\lg N$	6.667 $\lg N$
C_5	0.774 $\lg N$	3.869 $\lg N$
double-exchange	0.75 $\lg N$	3.75 $\lg N$
<u>Unbounded Degree</u>		
binary hypercube	1 $\lg N$	1 $\lg 2N$
mesh-connected hypercube	variable	0.742 $\lg 2N$
complete hypercube	variable	0.742 $\lg N$

3.1. Properties of the Homomorphism

A graph homomorphism is a mapping that preserves adjacency. For a given n , define a function h from all binary strings of length n to the even-parity binary strings of length n by $h(v_0v_1\dots v_{n-1}) \rightarrow w_0v_1\dots v_{n-1}$, where $w_i = v_i \oplus v_{i+1}$. Intuitively, this function identifies each vertex v of a shuffle-exchange graph with its complementary vertex $\bar{v}$.

Theorem 1. The mapping h is a homomorphism from a shuffle-exchange graph with 2^n vertices onto an even double-exchange graph with 2^{n-1} vertices.

Proof. It is clearly onto, mapping two strings v to each even-parity string w : v_0 may be chosen arbitrarily, and for $i > 0$, $v_i = v_0 + \sum \{w_j | 0 \leq j < i\}$. We call the string where v_0 is zero, w^0 , and the other w^1 .

The mapping h preserves adjacency. If $u = p(v)$ or $p^{-1}(v)$, the fact that the subscripts are calculated modulo n immediately ensures that $h(u) = p(h(v))$ or $p^{-1}(h(v))$, respectively. If $u = SE(v) = v_0v_1\dots v_{n-2}\bar{v}_{n-1}$, $h(u) = v_0\oplus v_1 v_1\oplus v_2 \dots v_{n-3}\oplus v_{n-2} \bar{v}_{n-2}\oplus v_{n-1} \bar{v}_{n-1}\oplus v_0 = v_0\oplus v_1 v_1\oplus v_2 \dots v_{n-3}\oplus v_{n-2} \bar{v}_{n-2}\oplus v_{n-1} \bar{v}_{n-1} \bar{v}_{n-1}\oplus v_0 = DE(h(v))$.

□

Corollary 1.1 $h(u) = h(v)$ if and only if $u = v$ or $u = \bar{v}$.

Proof. The "if" follows from the definition of h ; the "only if" is established by a straightforward induction on the bits of u and v .

This homomorphism means that any distributed algorithm running on a shuffle-exchange with N processors can be emulated by an even double-exchange with $N/2$ processors, with each processor in the even double-exchange emulating exactly two processors in the shuffle-exchange. (In Fishburn's terminology [50], the emulation is computationally uniform.) The even double-exchange has $3/4$ the diameter of the shuffle-exchange, so that algorithms that depend on broadcasting or echoing [6] should run in less than twice the time with half as many processors. The lower diameter and average distance can also reduce the average traffic load per processor, reducing queuing delays and congestion. The next theorem shows that most edges in the even double-exchange correspond to exactly two edges in the shuffle-exchange. This result suggests that emulation should not worsen traffic congestion, and also allows us to establish bounds on the VLSI layout area required by even double-exchanges.

Theorem 2. Each edge in the even double-exchange is the image of exactly two edges in the shuffle-exchange, except for the edge $(1^0, 1^0 01)$ when n is odd and the edge $((10)^*, (01)^*)$ when n is a multiple of four. These two edges are each the image of four edges in the shuffle-exchange.

Proof. Each edge (u, v) in the even double-exchange is the image of at least two edges in the shuffle-exchange.

If $v = p(u)$, the edge is an image of the edges $(u^0, p(u^0))$ and $(u^1, p(u^1))$, and similarly for p^{-1} ; if $v = DE(u)$, the edge is an image of the edges $(u^0, SE(u^0))$ and $(u^1, SE(u^1))$. If h maps any other edge in the shuffle-exchange graph to (u, v) , u^0 must obey one of these three conditions:

- a) $p(u^0) = \neg(SE(u^0))$,
- b) $p^{-1}(u^0) = \neg(SE(u^0))$,
- c) $p(u^0) = \neg(p^{-1}(u^0))$.

Condition a is only satisfied when n is odd and u^0 has the form $(01)^*0$; condition b is only satisfied when n is odd and u^0 has the form $(01)^*1$. In these cases, either u is 1^0 and v is $1^0 01$, or vice versa. Condition c is only satisfied when n is a multiple of four and u^0 has the form $(0011)^*$ or $(0101)^*$. In both cases, either u is $(10)^*$ and v is $(01)^*$, or vice versa.

□

3.2. VLSI Layout of the Even Double-Exchange

If advances in VLSI and wafer-scale integration permit entire networks of simple processors to be on one chip or wafer, the additional criterion of layout area may become important. Our model for the area required for the VLSI layout of a graph is due to Thompson [51,52]. In this model, processors and lines are laid out on a uniform rectilinear grid, with processors only being placed on grid intersections and lines only being allowed to follow the grid. (In other words, a processor must have integral coordinates and all points in a line must have at least one integral coordinate.) A line may not cross a processor, and is terminated by a processor at each end. Two lines may not occupy the same path in the grid, and may cross only at a grid intersection. The area of a layout is then the area of the smallest rectangle that contains all processors and lines. Among trivalent graphs, both the cube-connected-cycles and the shuffle-exchange require $\Theta(n^2/\lg^2 n)$ area [52,27,4]. We will use these results and the homomorphism h to show that the even double-exchange also requires $\Omega(n^2/\lg^2 n)$ area, and can be laid out in $O(n^2/\lg^3 n)$ area.

Definitions. A bisection of a graph $G = (V, E)$ is a subset E_B of E and a partition of V into two sets V_1 and V_2 such that $|V_1|$ is within one of $|V_2|$ and E_B contains each edge connecting a vertex in V_1 with a vertex in V_2 . The width of a bisection is $|E_B|$; the minimum bisection width of a graph is the least width of any bisection of the graph.

Theorem 3. If there is a bisection of an even double-exchange graph of width m , then there is a bisection of the corresponding shuffle-exchange graph of width either $2m$ or $2m+2$.

Proof. An even double-exchange graph always has an even number of vertices, so any bisection must have $|V_1| = |V_2|$. If E_B has m edges, then by Theorem 2, $h^{-1}(E_B)$ has either $2m$ or $2m+2$ edges; $h^{-1}(V_1)$ and $h^{-1}(V_2)$ partition the vertices of the shuffle-exchange, and, by Theorem 1, have equal cardinality. Furthermore, $h^{-1}(E_B)$ must include any edge of the shuffle-exchange between a vertex v in $h^{-1}(V_1)$ and a vertex w in $h^{-1}(V_2)$, since the edge $(h(v), h(w))$ connects a member of V_1 with a member of V_2 in the even double-exchange. Thus, the inverse images of E_B , V_1 , and V_2 provide a bisection of the corresponding shuffle-exchange, $h^{-1}(G)$.

□

Corollary 3.1. The minimum bisection width of an even double-exchange graph is at least $\Omega(N/\lg N)$.

Proof. By Theorem 3, if the even double-exchange had a smaller bisection width, then so would the shuffle-exchange. Thompson, however, proved that the minimum bisection width of the shuffle-exchange is $\Omega(N/\lg N)$.

□

Corollary 3.2. Any VLSI layout of the even double-exchange family requires area $\Omega(N^2/\lg^2 N)$.

Proof. Thompson proved that the VLSI layout of a graph requires at least the square of its minimum bisection width.

□

Any set of m edges can be added to a layout by adding no more than m vertical and m horizontal tracks, increasing the area by $O(m^2)$. Together with Corollary 3.2, this fact simplifies the description of proposed layouts for the even double-exchange family, by allowing any particular set of $O(N/\lg N)$ edges or vertices to be ignored without invalidating the proof of the final area requirements.

Theorem 4. The even double-exchange family has a VLSI layout with $O(N^2/\lg^3 N)$ area.

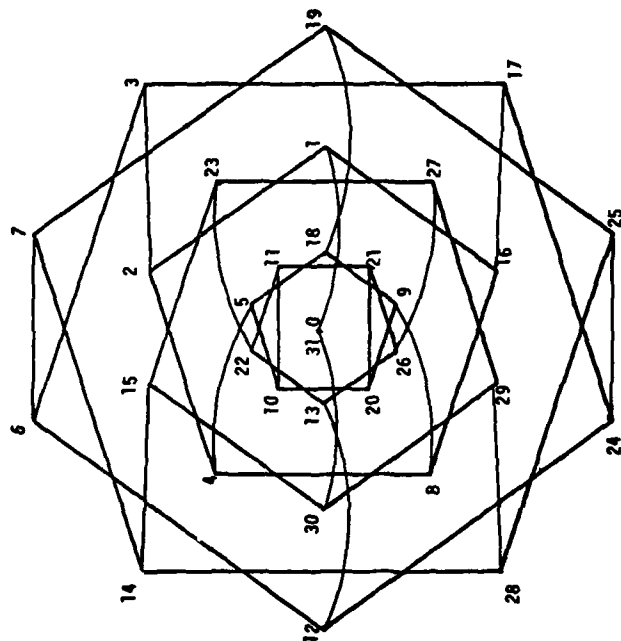
Proof. A necklace is a cycle of addresses created by successive p operations; each p edge and each vertex in a shuffle-exchange or even double-exchange graph lies in exactly one necklace, with at most $n = \lg N$ members. The length of any necklace must divide n , so at most $N^{1/2} \lg N$ vertices belong to necklaces with fewer than n members; this result in turn implies that there are $O(N/\lg N)$ necklaces. For the remainder of the proof, we assume all necklaces have n members, as $N^{1/2} \lg N$ is $O(N/\lg N)$.

Hoey and Leiserson [3] defined a layout of the shuffle-exchange based on a mapping g of the n -bit binary strings into the complex plane:

$$g(x_0 x_1 \dots x_{n-1}) = \sum_{0 \leq j < n} x_j z^{n-1-j},$$

where $z = e^{2\pi i/n}$, the principal primitive complex n -th root of 1. Hoey and Leiserson prove that $g(p(x))$ in the complex plane is $g(x)$ rotated counterclockwise around the origin by $2\pi/n$ radians, so all vertices in a necklace are the same distance from the origin, and that $g(SE(x))$ is $g(x)$ plus or minus 1, so all SE edges connect vertices at the same distance from the real axis. The shuffle-exchange graph for $n = 5$, as it appears

Figure 3.1



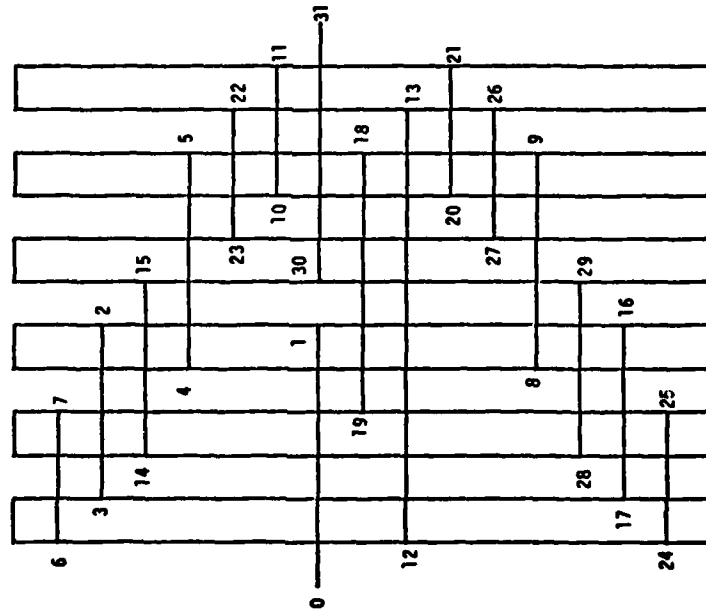
Mapping of the Shuffle-Exchange to the Complex Plane, for $n = 5$

under the mapping g , is illustrated in Figure 3.1 (from [3]).

To produce a layout for the shuffle-exchange, each necklace is given two adjacent vertical tracks for its vertices: the righthand track holds the vertices from the right half of the necklace's representation under g , and the left track holds the left half. All p edges are thus routed using a total of $O(N/\lg N)$ vertical tracks, together with two horizontal tracks to connect the two halves of each necklace. Because the vertices are placed on the vertical tracks according to their distance from the x -axis under g , every SE edge lies in a single horizontal track. Kleitman, Leighton, Lepley, and Miller [4] show that when the necklaces are ordered along the x -axis according to their distance in the complex plane from the origin, only $O(N/\lg^{1/2} N)$ horizontal tracks are needed to hold all SE edges, so the Hoey and Leiserson layout has $O(N^2/\lg^{3/2} N)$ area. Figure 3.2 shows the resulting layout for $n = 5$.

We may now use the homomorphism h to define an $O(N^2/\lg^{3/2} N)$ area layout for the even double-exchange family. There are at most $O(N/\lg N)$ addresses that g maps to the origin, so we can ignore all x such that $g(h^{-1}(x)) = \{0\}$. For all other x , we define $\text{Model}(x)$ to be $y \in h^{-1}(x)$ such that $\text{Im}(g(y)) > 0$ or $\text{Im}(g(y)) = 0$ and

Figure 3.2

Layout of the Shuffle-Exchange, for $n = 5$

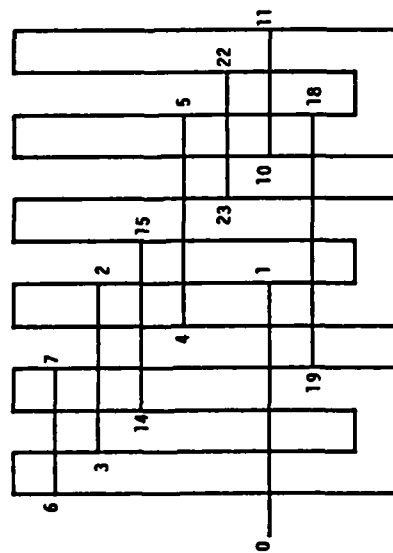
$\operatorname{Re}(g(y)) > 0$. Because $g(x) = -g(z)$, $\operatorname{Model}(x)$ is single-valued. We now place each vertex x of the even double-exchange according to the position of $\operatorname{Model}(x)$ in the shuffle-exchange layout.

If $\operatorname{Model}(x)$ and $\operatorname{Model}(p(x))$ lie in the same necklace of the shuffle-exchange, we connect x and $p(x)$ with the same layout routing. In the complex plane representation of the shuffle-exchange, each necklace has only two p edges connecting vertices below the real line with vertices on or above it, so at most two p edges in each even double-exchange necklace will not be handled by using the corresponding shuffle-exchange layout routing. These $O(N/\lg N)$ edges can be ignored, as usual. Similarly, we can connect x with $\operatorname{DE}(x)$ by using the layout routing for the edge between $\operatorname{Model}(x)$ and $\operatorname{Model}(\operatorname{DE}(x))$; the only possible exceptions arise when $\operatorname{SE}(\operatorname{Model}(x))$ lies on the real axis and has a negative imaginary component. Each necklace in the shuffle-exchange has at most two vertices whose images under g are on the real axis, so, once again, there are at most $O(N/\lg N)$ DE edges in the even double-exchange that this layout does not handle. Thus, all but $O(N/\lg N)$ edges in the even double-exchange can be routed within area $O(N^2/\lg^3 2N)$ using the Model mapping and the shuffle-exchange routings. Figure 3.3 shows the even double-exchange layout

for $n = 5$ corresponding to the shuffle-exchange layout of Figure 3.2.

We conjecture that Kleitman's $O(N^2/\lg^2 N)$ layout of the shuffle-exchange may similarly be used to provide an optimal layout of the even double-exchange; in any case, ad hoc layouts for practical values of N can easily improve on these general constructions for the shuffle-exchange as well as the even double-exchange.

Figure 3.3

Layout of the Even Double-Exchange, for $n = 5$

3.3. Summary

The diameter of the even double-exchange is qualitatively superior to the shuffle-exchange and all other trivalent families previously proposed, and yet this family can be efficiently laid out and effectively used to emulate algorithms proposed for the shuffle-exchange. In the next chapter, we show that the even double-exchange is at least as reliable as the shuffle-exchange.

Chapter 4 Reliability

A multicomputer graph might be highly successful in terms of the cost measures considered in Chapter 2 yet still be impractical. Any pragmatic comparison of multicomputer graph families must measure their reliability: How seriously do failures affect them? Of the three major effects of failures -- disconnection, increased diameter, and increased traffic congestion -- this chapter concentrates on the most serious, disconnection. We are deliberately excluding the issue of fault diagnosis, so the measures presented here assume that failed elements are identified by some underlying protocol.

Knowing that a graph remains connected despite some number of failures, however, is not an adequate measure of reliability for a distributed computer. Even if the graph is still connected, we need to know if messages can be routed using local knowledge, or if their routing requires global knowledge of the remaining network. Global knowledge is a particularly onerous requirement for distributed computers because extensive exchanges of routing information increase the congestion on a network already suffering from failures. Local rerouting

methods for handling failures allow us, in turn, to establish lower bounds for a general reliability measure, the Average Random Failset size: If failures are randomly and uniformly distributed without replacement, how many can be tolerated on the average before the graph becomes disconnected?

4.1. Connectivity

In graph-theoretic analyses, a common measure of reliability [53,7,54,9] is connectivity: the least number of failures that can disconnect the network. For edge-connectivity, the failures are of edges; for vertex-connectivity (or simply 'connectivity'), the failures are of vertices. Vertex-connectivity is sometimes also called survivability [55,56] or invulnerability [57,54]. By elementary graph theory, the vertex-connectivity is bounded above by the edge-connectivity, which in turn is bounded above by the smallest degree of any vertex. Connectivity provides a worst-case bound for failures; most of the proposed families have their vertex-connectivity equal to their minimum degree, and therefore equal to their edge-connectivity.

4.1.1. Previous multicomputer graphs

4.1.1.1. Graph product families

The graph product families generally have high connectivity, in compensation for their poor performance in respect to degree and cost. Given two graphs G_1 and G_2 with vertex-connectivities c_1 and c_2 respectively, the connectivity of their Cartesian product $G_1 \times G_2$ is greater than or equal to $\min(c_1 + c_2, c_1 n_2, c_2 n_1)$. When the

operand graphs' connectivities are as high as possible, with $c_1 = d_1$ and $c_2 = d_2$, the product's connectivity is bounded above by $c_1 + c_2$ (the degree of the product). This result shows that the connectivity and the edge-connectivity of the binary, the complete, and the mesh-connected hypercubes are exactly their degree.

4.1.1.2. Tree variations

Trees are, of course, 1-connected, while Friedman's joined trees of degree d are d -connected. Trivalent Multi-Tree-Structures are 3-connected; for higher degrees, Multi-Tree-Structure graphs may have a connectivity as low as three, depending on the details of their construction. Hypertrees always have a root vertex of degree 2, and so are no better than 2-connected at any degree. Goodman and Sequin prove that Hypertree 1 has an edge-connectivity of 2, by showing how to route around a single failed edge. More generally, all hypertrees are proven 2-connected by noting that there is always a route around any single failed vertex: If the vertex is the root, its two children are always connected by a hypercube-like edge; if the vertex is a leaf, all other vertices are always connected by the underlying binary tree; otherwise, each of the two children of the failed vertex always has a hypercube-like edge to a

vertex that was not a child of the failed vertex, from which all vertices outside the failed vertex's subtree can be reached (including the vertex connected to the other child, so its subtree is still accessible.)

4.1.1.3. Single-exchange graphs

Connectivity varies widely among single-exchange families. At degree three, the cube-connected-cycles have connectivity 3, while the shuffle-exchange graphs always have two vertices of degree 1, and so are only 1-connected. At degree four, the binary de Bruijn networks have two vertices of degree 2, so cannot be more than 2-connected; the rerouting method defined in a later section proves that they are, in fact, 2-connected. Saluja and Reddy [58] have modified the binary de Bruijn to create a 4-connected family with, however, each graph having two vertices of degree 5. In general, de Bruijn networks with base b and degree $d = 2b$ are $(d-2)$ -connected. The lens with any degree $d = 2b$ is d -connected, provided n is at least 3, as is the multistage de Bruijn.

4.1.2. Connectivity of the Moebius graph

The Moebius graphs can be modified to make them 3-connected, while preserving their degree, diameter, in-

terpolation, and routing properties. No graph with degree 3 can have higher connectivity, so, by this measure of reliability, these modifications make the Moebius family as reliable as possible without increasing the degree. Throughout this section, ρ denotes the twisted rotate operation, $\rho(v_0 v_1 \dots v_{n-1}) = v_1 v_2 \dots v_{n-1} v_0$.

The minimum degree of vertices in the unmodified Moebius graphs is two, setting an upper bound on their connectivity. However, very few vertices in a given Moebius graph have so few edges.

Lemma 1.1. Given a Moebius graph with $N = 2^n$ vertices ($n > 1$), if n is even, all vertices have degree three; if n is odd, all vertices have degree 3 except the vertex with address $\lfloor N/3 \rfloor$ and the vertex with the complementary address, which have degree 2 and are connected to one another.

Proof. There are three operations that define edges in a Moebius: DE , ρ , and ρ^{-1} . Matching corresponding bits between x and $DE(x)$ immediately yields the contradiction that $x_0 = \bar{x}_0$, so that $DE(x)$ cannot equal x ; similar arguments show that $\rho(x)$ and $\rho^{-1}(x)$ also cannot equal x , and that $DE(x)$ is never equal to $\rho(x)$ or $\rho^{-1}(x)$. These results prove that every vertex in a Moebius graph has degree at least two. Suppose that $\rho^{-1}(x) = \rho(x)$.

Matching bits as before gives the equations $x_1 = \bar{x}_{n-1}$, $x_0 = \bar{x}_{n-2}$, and $\forall i < n-2, x_i = x_{i+2}$. If n is even, these equations are inconsistent; if n is odd, there are two solutions: $x = \lfloor N/3 \rfloor = (01)^*0$ and $x = (10)^*1$. These two addresses are connected to each other by ρ ; all other vertices cannot have $\rho^{-1} = \rho$, and so are trivalent.

M

Because so few vertices in a Moebius graph have degree less than 3, it is easy to modify the Moebius construction to produce regular trivalent graphs that are 3-connected: If n is odd, delete the two connected vertices of degree two and create an edge connecting the two vertices that were adjacent to them. We call this operation eliding the degree-2 vertices. The Moebius routing algorithm still works for the elided Moebius, and the diameter bound is unharmed, since any path through the elided vertices is shortened. In particular, the two vertices that once neighbored on the elided vertices do not need to make any change in their message forwarding: A message that would have gone through one of the elided vertices must get there on a DE edge, and, if sent out on the prescribed DE edge reaches exactly the vertex that lies two steps farther along its calculated path.

Theorem 1. The elided Moebius graph is 3-connected.

For $n < 5$, exhaustive testing shows that the graph is 3-connected; for $n \geq 5$, we prove Theorem 1 by defining some sets of distinguished vertices such that despite any two vertex failures, there is a path from any nondistinguished vertex to some surviving distinguished vertex (Lemma 1.2), and a path between any two surviving distinguished vertices (Lemma 1.2).

Definitions. The proof employs three sets of distinguished vertices, Zero, One, and Altern:

$$\begin{aligned} \text{Zero} &= \{0n\} \\ \text{One} &= \{1n\} \\ \text{Altern} &= \{(01)n/2, (10)n/2\} \text{ for } n \text{ even} \\ &\quad \{(01)(n-3)/2001, (01)(n-1)/21, \\ &\quad \quad (10)(n-1)/20, (10)(n-3)/2110\} \text{ for } n \text{ odd} \end{aligned}$$

The names Zero, One, and Altern are used both for the sets and for their members. An address not in Zero, One, or Altern is called nondistinguished.

Let $x = x_0x_1 \dots x_{n-1}$, and define InitialZero(x) to be the number of leading zeros in x . InitialZero(x) is thus the least i such that $x_i = 1$, or n if x is Zero. Similarly, InitialOne(x) denotes the number of leading ones in x , and InitialAltern(x) denotes the length of the longest initial string of alternating bits. The following elementary properties of these initial pattern

lengths are used in the proofs below:

- a) $\text{InitialZero}(x) > 0$ if and only if $\text{InitialOne}(x) = 0$;
- b) $\text{InitialAltern}(x) > 1$ if and only if $\text{InitialZero}(x) \leq 1$ and $\text{InitialOne}(x) \leq 1$;
- c) If x and y differ in their values of InitialZero , or of InitialOne , or of InitialAltern , then $x \neq y$.

We occasionally use the analogous properties for the number of trailing zeros, ones, or alternating bits in an address x , denoted $\text{FinalZero}(x)$, $\text{FinalOne}(x)$, and $\text{FinalAltern}(x)$ respectively.

Lemma 1.2. For any nondistinguished vertex x there are vertex-disjoint paths from x to Zero, One, and some vertex $A \in \text{Altern}$.

Proof. Lemmas 1.2.1, 1.2.2, and 1.2.3 will establish that, from any nondistinguished x , there are paths $P_0(x)$, $P_1(x)$, and $P_A(x)$, to Zero, One, and some A in Altern , respectively, which monotonically increase in their vertices' values of InitialZero , InitialOne , and InitialAltern , respectively. Although these paths eventually acquire sufficiently long initial bit patterns to guarantee the disjointness of their remaining vertices, they do not suffice to prove Lemma 1.2; for example, if $x = 111101$, then both $P_0(x)$ and $P_1(x)$ include the vertex

111110.

Lemma 1.2.4 will circumvent this problem (and complete the proof) by specifying initial paths for each x , denoted $\text{InnerPath}_0(x)$, $\text{InnerPath}_1(x)$, and $\text{InnerPath}_A(x)$, to vertices $\text{StartP}_0(x)$, $\text{StartP}_1(x)$, and $\text{StartP}_A(x)$ whose values of InitialZero , InitialOne , and InitialAltern , respectively, are large enough to keep the outer paths $P_0(\text{StartP}_0(x))$, $P_1(\text{StartP}_1(x))$, and $P_A(\text{StartP}_A(x))$ disjoint from one another and from the inner paths. The inner paths will also be constrained to be vertex-disjoint; the outer paths will be disjoint and will contain no elided vertices because they will have, respectively, InitialZero , InitialOne , and InitialAltern greater than one.

Lemma 1.2.1. Given any nondistinguished vertex x , there is a path, $P_0(x)$, to Zero whose vertices increase monotonically in their InitialZero values.

Proof. There are two cases to consider, depending on the value of $\text{InitialZero}(x)$. If $\text{InitialZero}(x)$ is less than $n-2$, we can route to an address y such that $\text{InitialZero}(y) = \text{InitialZero}(x) + 1$: If x ends with a one, let y be $p^{-1}(x)$; otherwise, let y be $p^{-1}(\text{DE}(x))$. Since $\text{InitialZero}(x) < n-2$ the DE does not change $\text{InitialZero}(x)$, while the p^{-1} increases $\text{InitialZero}(x)$

by exactly 1. Applying this construction a finite number of times must reach an address whose InitialZero value is at least $n-2$, along a path whose InitialZero values increase monotonically.

If InitialZero(x) is at least $n-2$, we can demonstrate a path from x to Zero whose vertices increase monotonically in their value of InitialZero. All bits to the left of x_{n-2} must be 0, so there are four cases to consider. If the rightmost two bits are both zero, x is Zero. If the rightmost two bits are both one, x 's DE edge connects directly to Zero. If the rightmost two bits are 10, a DE changes them to 01, increasing the InitialZero value by one and reducing to the fourth case: If the rightmost two bits are 01, x 's p^{-1} edge connects directly to Zero.

M

We note that if InitialZero(x) is greater than one, $P_0(x)$ can contain no elided vertices because InitialAltern will be 1 for the entire path.

Lemma 1.2.2. Given any nondistinguished vertex x , there is a path, $P_1(x)$, to One whose vertices increase monotonically in their InitialOne values.

Proof. The proof exactly parallels that of Lemma 1.2.1.

M

If InitialOne(x) is greater than one, $P_1(x)$ can contain no elided vertices.

Lemma 1.2.3. Given any nondistinguished vertex x , there is a path, $P_A(x)$, to some $A \in \text{Altern}$ whose vertices increase monotonically in their InitialAltern values.

Proof. For InitialAltern(x) $< n-2$ or n even, the proof exactly parallels that of Lemma 1.2.1. When n is odd, the only nondistinguished vertices with InitialAltern $\geq n-2$ are $(01)^{n-2}000$ and $(10)^{n-2}111$, from each of which a DE edge increases the value of InitialAltern by one and connects to a member of Altern.

M

Because InitialAltern increases by no more than one at each step, $P_A(x)$ could only reach an elided vertex by first passing through a vertex with InitialAltern = $n-1$; when n is odd, both such vertices are members of Altern, so $P_A(x)$ ends with them.

Lemma 1.2.4. Given any nondistinguished vertex x , there are vertices Start $P_0(x)$, Start $P_1(x)$, and Start $P_A(x)$, and paths InnerPath $_0(x)$, InnerPath $_1(x)$, and InnerPath $_A(x)$ reaching Start $P_0(x)$, Start $P_1(x)$, and Start $P_A(x)$ respectively, which satisfy three properties:

- a) the addresses in the proposed paths do not represent elided vertices;
- b) the inner paths are vertex-disjoint;
- c) $\text{InnerPath}_0(x)$ does not intersect $P_1(\text{StartP}_1(x))$ or $P_A(\text{StartP}_A(x))$; $\text{InnerPath}_1(x)$ does not intersect $P_0(\text{StartP}_0(x))$ or $P_A(\text{StartP}_A(x))$; and $\text{InnerPath}_A(x)$ does not intersect $P_0(\text{StartP}_0(x))$ or $P_1(\text{StartP}_1(x))$.

Proof. Unfortunately, there are numerous cases, distinguished by the initial and final bit patterns of x , that must be proven separately. Only addresses beginning with 0 will be discussed; those beginning with 1 are handled by complementing all addresses. The detailed verification of the properties is very similar for each case, and so is presented for only one; correct inner paths for all cases are listed in Table 4.1.

Case $\text{InitialZero}(x) > 1$ and x ends in 10

$$\text{InnerPath}_0 = DE$$

$$\text{InnerPath}_1 = p^{-1} DE p^{-1}$$

$$\text{InnerPath}_A = p DE p^{-2}$$

The vertex x must have the form $00x_2 \dots x_{n-3}10$, with at least one bit between the 00 and the 10. InnerPath_0 contains only $\text{StartP}_0 = 00x_2 \dots 01$. Neither it nor the vertices in $P_0(\text{StartP}_0)$ are elided because they have

$\text{InitialZero} \geq \text{InitialZero}(x) \geq 2$. InnerPath_1 contains $10x_2 \dots 11$, $100x_2 \dots 0$, and $\text{StartP}_1 = 1100x_2 \dots$. These addresses have $\text{InitialAltern} \leq 2$, and so cannot have been elided; they have $\text{InitialOne} \geq 1$, and so are not in InnerPath_0 or $P_0(\text{StartP}_0)$. The vertices in $P_1(\text{StartP}_1)$ have $\text{InitialOne} \geq 2$ and therefore do not include any members of InnerPath_0 .

InnerPath_A contains $0x_2 \dots 101$, $0x_2 \dots 110$, $10x_2 \dots 11$, and $\text{StartP}_A = 010x_2 \dots 1$. Since x is not elided, $p(x)$ cannot be; the other vertices are not elided because they have $\text{FinalAltern} \leq 2$ when n is odd. The addresses in InnerPath_A are distinct from those of InnerPath_0 and $P_0(\text{StartP}_0)$ because they have $\text{InitialZero} < \text{InitialZero}(x)$, and are distinct from those of $P_1(\text{StartP}_1)$ because they have $\text{InitialOne} < 2$. Only $10x_2 \dots 11$ could possibly be in InnerPath_1 , but it begins with $10^{\text{InitialZero}(x)-1}1$, while each address in InnerPath_1 starts with either $10^{\text{InitialZero}(x)}1$ or 11 . Finally, each address in $P_A(\text{StartP}_A)$ has $\text{InitialAltern} \geq 3$, and so cannot belong to InnerPath_0 or InnerPath_1 .

Analogous arguments establish the three necessary conditions for the inner paths for all the other cases, completing the proofs of lemmas 1.2.4 and 1.1.

■

Table 4.1

InitialAltern(x) = n-1
 If n is even, x must be (01) (n-2)/200.

InnerPath₀ = DE p⁻²
 InnerPath₁ = p² DE p⁻²
 InnerPath_A = p⁻¹

If n is odd, x is distinguished.

InitialAltern(x) = n-2

If n is even

If x = (01) (n-2)/201, ⁻¹
 InnerPath₀ = p² DE p⁻¹
 InnerPath₁ = p⁻¹ DE p⁻¹
 InnerPath_A = DE

If x = (01) (n-2)/211,

InnerPath₀ = p⁻¹
 InnerPath₁ = p DE p⁻¹
 InnerPath_A = DE p⁻¹

If n is odd

If x = (01) (n-3)/2000,
 InnerPath₀ = DE p⁻¹
 InnerPath₁ = p DE p⁻¹
 InnerPath_A = p⁻¹

If x = (01) (n-3)/2001, it's distinguished.

InitialAltern(x) > 1 but < n-2

If x ends in 0,

InnerPath₀ = DE p⁻¹
 InnerPath₁ = p DE p⁻¹
 InnerPath_A = p⁻¹

If x ends in 1,

InnerPath₀ = p⁻¹
 InnerPath₁ = p DE p⁻¹

Table 4.1 (continued)

InitialAltern(x) = 1 (so InitialZero(x) > 1)

If x ends in 00

If InitialZero(x) = n, x is distinguished
 If InitialZero(x) < n-2

InnerPath₀ = DE
 InnerPath₁ = p⁻²
 InnerPath_A = p DE p⁻¹

If x ends in 10

InnerPath₀ = DE
 InnerPath₁ = p⁻¹ DE p⁻¹
 InnerPath_A = p DE p⁻¹

If x ends in 01

If InitialZero(x) = n-1

InnerPath₀ = p⁻¹
 InnerPath₁ = p² DE p⁻²
 InnerPath_A = DE p⁻¹

If InitialZero(x) < n-2

InnerPath₀ = p⁻¹
 InnerPath₁ = DE p⁻¹ DE p⁻¹
 InnerPath_A = p DE p⁻¹ DE p⁻¹

If x ends in 11

If InitialZero(x) = n-2

InnerPath₀ = DE
 InnerPath₁ = p DE p⁻²
 InnerPath_A = p⁻¹ DE p⁻²

If InitialZero(x) < n-2

InnerPath₀ = p⁻¹
 InnerPath₁ = DE p⁻²
 InnerPath_A = p DE p⁻¹ DE p⁻¹

Lemma 1.3. Despite any two vertex failures, there is a path between any two surviving distinguished vertices.

Proof. The proof uses four sublemmas. We show that there are three vertex-disjoint paths between zero and one (Lemma 1.3.1), three between any member of Altern and zero (Lemma 1.3.2), and three between any member of Altern and one (Lemma 1.3.3). Finally, we show that despite any two vertex failures, there is a path between any two surviving members of Altern (Lemma 1.3.4).

Lemma 1.3.1. There are three vertex-disjoint paths between zero and one that contain no elided vertices.

Proof. Starting from zero, define paths P_L , P_R , and P_M :

$$\begin{aligned} P_L &= \rho^{-(n-2)} DE \\ P_R &= DE \rho^{n-2} \\ P_M &= (\rho DE)^{n-1} \rho^{-1} \end{aligned}$$

P_L introduces 1's from the left, P_R from the right, and P_M from the right, but with the rightmost two bits being either 01 or 10. Straightforward induction arguments exactly characterize all nonterminal vertices in these three paths: those in P_L have the form 11^*00^* , those in P_R have the form 00^*111^* , and those in P_M have the form of either 0^*1^*01 or 0^*1^*10 . These characterizations prove that the sets of nonterminal vertices of P_L , P_R , and P_M are pairwise disjoint and, since n is greater

that 4, contain no elided vertices.

$\square$

Lemma 1.3.2. There are three vertex-disjoint paths from any member of Altern to zero.

Proof. The required paths are defined using two convenient shorter paths:

$$\begin{aligned} \text{Rightshift}(x) &= \rho^2 DE \rho DE \rho, \\ \text{Leftshift}(x) &= DE \rho^{-1} DE \rho^{-3}. \end{aligned}$$

Letting a denote either 0 or 1, then for any x , $\text{Rightshift}(axaax) = xaaxa$ and $\text{Leftshift}(xaaxa) = aaxax$. In the addresses along the paths defined by $\text{Leftshift}(xaaxa)$ and $\text{Rightshift}(axaax)$, the substring x is shifted in position, but never changed. $\text{Leftshift}(xaaxa)$ also has the property that the number of initial a 's increases monotonically: $xaaxa$ goes to $xxaax$, $axxax$, $axxxa$, $axxxx$, $axxxx$, and $axxxx$ in turn.

For the remainder of this lemma, let q denote $n \text{ div } 4$ and r denote $n \text{ modulo } 4$. The particular paths between Altern and zero depend on r , but share a common pattern: P_L shifts in zeros from the left, P_R shifts in zeros from the right, and P_M first shifts in ones until the address is almost entirely ones and then shifts in zeros.

Case $r = 0$: Altern has two members, $(0101)^q$ and $(1010)^q$.

Starting from $(0101)^q$,

$$P_L = \rho^{-1} \text{Leftshift}^{q-1} \text{DE } \rho^{-1},$$

$$P_R = \rho \text{DE } \rho \text{Rightshift}^{q-1},$$

$$P_1 = \text{DE } \rho \text{DE } \rho^{-1} \text{DE } \rho^{-2} \text{Leftshift}^{q-1} \rho^{-(n-2)} \text{DE}.$$

The initial ρ^{-1} of P_L transforms $(0101)^q$ to $0010(1010)^{q-1}$. From here, the value of InitialZero increases monotonically. The final DE converts the last address of the last Leftshift, $(0000)^{q-1}0010$, to $0^{n-1}1$, which the ρ^{-1} takes directly to zero. None of these internal addresses can appear in P_R , because all addresses along that path have InitialAltern > 1 : P_R first reaches $(1010)^{q-1}1011$, $(1010)^{q-1}1000$, and $(0101)^{q-1}0000$; the next $q-2$ Rightshifts produce only addresses starting with 01 or 10, ending with $0101(0000)^{q-1}$; the final Rightshift then produces $101(0000)^{q-1}1$, $01(0000)^{q-1}10$, $01(0000)^{q-1}01$, $1(0000)^{q-1}011$, 10^{n-1} , and zero. Except for $0^{n-1}1$ in P_L , all addresses in P_L and P_R contain the pattern 10. This fact, together with their initial bit patterns, distinguishes them from most addresses in P_1 .

The path P_1 begins with $(0101)^{q-1}0110$, $1(0101)^{q-1}101$, $1(0101)^{q-1}110$, and $11(0101)^{q-1}11$. InitialOne increases monotonically from here until, after

the Leftshifts, the path reaches One. All addresses in this portion of P_1 have InitialZero < 2 , and so do not appear in P_L , while those with InitialOne > 1 must be distinct from all in P_R . The only addresses in this portion with InitialOne ≤ 1 are the first three, which clearly differ from the first two addresses in P_R , and are distinct from later addresses in P_R because the latter all contain a sequence of four or more zeros. After reaching One, P_1 employs ρ^{-1} s to produce addresses of the form 0^+1^+11 . These addresses do not contain the pattern 10, and so are distinct from all addresses in P_L and P_R . The last ρ^{-1} in P_1 produces $0^{n-2}11$, from which the concluding DE immediately reaches zero.

Thus all three proposed paths reach zero without intersecting, and the first case in the proof of Lemma 1.3.2 is established. The paths for $(1010)^q$ and for the remaining cases are listed in Table 4.2; their proofs are so similar that they are omitted.

Table 4.2

Case $r = 0$: Altern has two members
 Starting from $(0101)q$,
 $P_L = \rho^{-1} \text{Leftshift}^{q-1} \text{DE} \rho^{-1}$
 $P_R = \rho \text{DE} \rho \text{Rightshift}^{q-1}$
 $P_1 = \text{DE} \rho \text{DE} \rho^{-1} \text{DE} \text{Leftshift}^{q-1} \rho^{-(n-2)} \text{DE}$
 Starting from $(1010)q$,
 $P_L = \text{Leftshift}^{q-1} \text{DE} \rho^{-1} \text{DE} \rho^{-1} \text{DE}$
 $P_R = \rho \text{Rightshift}^{q-1} \rho^2 \text{DE} \rho$
 $P_1 = \rho^{-1} \text{Leftshift}^{q-1} \text{DE} \rho^{n-1}$

Case $r = 1$: Altern has four members
 Starting from $(0101)q$,
 $P_L = \rho^{-2} \text{Leftshift}^{q-1} \text{DE} \rho^{-1}$
 $P_R = \text{Rightshift}^q \rho$
 $P_1 = \text{DE} \rho^{-1} \text{Rightshift}^q \rho^{-(n-3)} \text{DE}$
 Starting from $(0101)q^{-1} 01001$,
 $P_L = \rho^{-4} \text{Leftshift}^{q-1} \text{DE} \rho^{-2} \text{DE}$
 $P_R = \text{Rightshift}^q \rho \text{DE} \rho^{-1}$
 $P_1 = \text{DE} \rho \text{DE} \text{Rightshift}^q \rho \text{DE} \rho^{n-1}$
 Starting from $(1010)q$,
 $P_L = \rho^{-1} \text{Leftshift}^q \rho \text{DE}$
 $P_R = \rho \text{Rightshift}^{q-1} \rho^2 \text{DE} \rho^{-1}$
 $P_1 = \text{DE} \text{Rightshift}^{q-1} \rho^2 \text{DE} \rho^{n-1}$
 Starting from $(1010)q^{-1} 10110$,
 $P_L = \rho^{-3} \text{Leftshift}^{q-1} \text{DE} \rho^{-1,2}$
 $P_R = \rho \text{Rightshift}^{q-1} \rho^2 \text{DE} \rho \text{DE}$

Table 4.2 (continued)

$P_1 = \text{DE} \rho \text{Rightshift}^q \rho^{n-2}$

Case $r = 2$: Altern has two members
 Starting from $(0101)q$,
 $P_L = \text{Leftshift}^q \rho^{-1}$
 $P_R = \rho \text{Rightshift}^q \rho^2$
 $P_1 = \rho^{-1} \text{Leftshift}^q \rho^{-n} \text{DE}$
 Starting from $(1010)q$,
 $P_L = \text{Leftshift}^q \text{DE} \rho^{-1}$
 $P_R = \rho \text{Rightshift}^q$
 $P_1 = \rho^{-1} \text{Leftshift}^q \rho^{-(n-2)} \text{DE}$

Case $r = 3$: Altern has four members
 Starting from $(0101)q$,
 $P_L = \rho^{-2} \text{Leftshift}^q \text{DE} \rho^{-3}$
 $P_R = \text{Rightshift}^q \rho^2 \text{DE} \rho$
 $P_1 = \text{DE} \text{Rightshift}^q \rho^2 \text{DE} \rho^{-(n-3)} \text{DE}$
 Starting from $(0101)q$,
 $P_L = \rho^{-2} \text{Leftshift}^q$
 $P_R = \text{Rightshift}^{q+1} \text{DE}$
 $P_1 = \text{DE} \rho \text{Rightshift}^{q+1} \rho^{n-2}$
 Starting from $(1010)q$,
 $P_L = \rho^{-1} \text{Leftshift}^q \text{DE} \rho^{-1,2}$
 $P_R = \rho \text{Rightshift}^q$
 $P_1 = \text{DE} \text{Rightshift}^q \rho^{-(n-2)} \text{DE}$
 Starting from $(1010)q$,
 $P_L = \rho^{-3} \text{Leftshift}^q \rho^{-1}$
 $P_R = \rho \text{Rightshift}^q$
 $P_1 = \text{DE} \rho \text{Rightshift}^q \rho^{-(n-3)} \text{DE}$

Lemma 1.3.3. There are three vertex-disjoint paths between any member of Altern and One.

Proof. The complement of any member of Altern is also in Altern, so complementing all addresses used in the proof of Lemma 1.3.2 produces the required paths from any member of Altern to One. M

Lemma 1.3.4. Despite any two vertex failures, there is a path between any two distinct members, A and B, of Altern.

Proof. Lemmas 1.3.2 and 1.3.3 have established that there are three vertex-disjoint paths between A and Zero, between A and One, between B and Zero, and between B and One. These paths ensure that, provided at least one of Zero and One has not failed, no other two failures can disconnect A from B. Lemma 1.3.4 is proven, then, if there is at least one path between A and B that goes through neither Zero nor One. If n is even, A and B must be $(01)^*1$ and $(10)^*1$, and the requisite path is produced by applying p n times; each vertex on this path has either the form $(10)^*1(10)^*1$ or the form $(01)^*(10)^*$, which guarantees that none is Zero or One for $n > 2$.

If n is odd, Altern has four members; for brevity, we name them by their final two bits: $\lambda_{00} = (10)^*0$, λ_{01}

$= (01)^*001$, $\lambda_{10} = (10)^*110$, and $\lambda_{11} = (01)^*1$. The elided Moebius has an edge directly connecting λ_{01} with λ_{10} . λ_{11} connects to λ_{00} through a single vertex: $p(\lambda_{11}) = (10)^*111$, and $DE((10)^*111) = \lambda_{00}$. Because n is greater than 3, the middle vertex on this path is neither Zero nor One. Finally, the path $DE p (DE p^2)^{(n-3)/2}$ connects λ_{00} to λ_{10} without going through Zero or One. Each vertex on this path has one of the three forms $(10)^*1(10)^*11$, $(01)^*(10)^*110$, or $(01)^*(10)^*1$. Because n is greater than 3, no vertex is Zero or One, and lemmas 1.3.4 and 1.3 are proven.

M

Proof of Theorem 1.

We can now complete the proof of Theorem 1, that the elided Moebius is 3-connected, by combining Lemmas 1.2 and 1.3. Despite any two failed vertices, there is a path between any two vertices v and w, which runs from v to some distinguished vertex D_v , from D_v to some distinguished vertex D_w , and from D_w to w. (Of course, not all of these formal vertices need actually be distinct.)

M

4.1.3. Connectivity of the shuffle-exchange

There are also several ways in which shuffle-exchange graphs can be modified to be 3-connected, without increasing their degree; the elided shuffle-exchange family presented here satisfies an additional reliability criterion, vertex reroutability, defined in the next section. Once self-loops and duplicate edges are eliminated, shuffle-exchange graphs have two vertices of degree 1, 0^* and 1^* , and, if n is even, two adjacent vertices of degree two, $(01)^*$ and $(10)^*$. To make the graphs 3-connected, we delete 0^* and 1^* , reducing the degree of 0^* and 1^* to two. We replace these vertices in turn by edges connecting 10^* to 0^* and 01^* to 1^* . If n is even, we also elide $(01)^*$ and $(10)^*$, making an edge from $(01)^*00$ to $(10)^*11$. (Thus we reduce N by 4 when n is odd and by 6 when n is even.) For $n > 3$, these elisions reduce the diameter while preserving the degree and the routing algorithms; using the idea of distinguished vertices and monotonic paths again, a case-by-case argument proves that these graphs are 3-connected. Zero is now the set $\{0^*10\}$, and One is $\{1^*01\}$; if n is odd, Altern is $\{(01)^*0, (10)^*1\}$, while if n is even, Altern is $\{(01)^*00, (10)^*11\}$. As the proof so closely parallels the Moebius proof, its details are omitted.

4.1.4. Connectivity of the even double-exchange

The even double-exchange graphs discussed in chapter 3 are only 1-connected, because each contains the degree-1 address 0^* . The technique of deleting vertices of degree 1 and eliding vertices of degree 2 again produces a family that is both 3-connected and vertex reroutable. The elisions required, and the particular distinguished vertices used to prove 3-connectness, depend on n modulo 4.

When n modulo 4 is 0, we remove the two vertices of degree 1, 0^* and 1^* , and elide their neighbors, 0^*11 and 1^*00 (thereby connecting 0^*110 with 10^*1 and 1^*001 with 01^*0), as well as eliding the two degree 2 vertices, $(01)^*$ and $(10)^*$ (producing an edge from $(01)^*10$ to $(10)^*01$). As long as n is at least 8, this produces a 3-connected graph; we set Zero = 0^*110 , One = 1^*001 , and Altern = $(01)^*10$. When n modulo 4 is 2, the only the vertices with degree below three are 0^* and 1^* , so we remove them and elide their neighbors, again connecting 0^*110 with 10^*1 and 1^*001 with 01^*0 . In this case, if n is at least 6, we can use Zero = 0^*110 , One = 1^*001 , and Altern = $\{(01)^*00, (10)^*11\}$. Finally, when n is odd and at least 5, we remove the single vertex of degree 1, 0^* , and elide its neighbor 0^*11 , to connect 0^*110 with 10^*1 . The only degree-2 vertices are 1^*0 and its neighbor

4.2. Local Rerouting

The 3-connectedness of the elided Moebius and shuffle-exchange families proves that they satisfy a popular measure of reliability for multicomputer graphs without, however, guaranteeing that even a single failure can be handled in an actual distributed system. The proofs employ very impractical message routes, several times longer than those of simple routing algorithms and with a severe bottleneck problem from the distinguished vertices. Instead of relying on distinguished vertices or universal knowledge of a failure, a realistic distributed routing algorithm should be able to reroute locally around an isolated failure. Our formal model for the cost of local rerouting turns out to be useful not only for providing a measure of the reliability of a distributed computer, but also for bounding the diameter costs of failures in multicomputer graphs and for determining an analytic lower bound on a general reliability measure, the Average Random Failset size.

4.2.1. Definitions

Given a graph $G = [V, E]$ that is at least 2-connected, it is always possible to reroute around an isolated failure. The cost of doing such rerouting locally will be measured by the minimum number of vertices

1^*01 , which we elide to make an edge from 01^* to 1^*011 . Here, Zero = 0^*110 and One = 1^*000 ; Altern is $(01)^*0$ when n modulo 4 is 1, and $(10)^*1$ otherwise.

As before, monotonic paths to Zero, One, and Altern are easily demonstrated, and a case-by-case argument then proves that each elided even double-exchange graph is 3-connected, for n at least 5. Furthermore, this elision does not increase the asymptotic space requirements of the layouts discussed in Chapter 3, because only a fixed number of vertices and edges are affected.

4.2. Local Rerouting

The 3-connectedness of the elided Moebius and shuffle-exchange families proves that they satisfy a popular measure of reliability for multicomputer graphs without, however, guaranteeing that even a single failure can be handled in an actual distributed system. The proofs employ very impractical message routes, several times longer than those of simple routing algorithms and with a severe bottleneck problem from the distinguished vertices. Instead of relying on distinguished vertices or universal knowledge of a failure, a realistic distributed routing algorithm should be able to reroute locally around an isolated failure. Our formal model for the cost of local rerouting turns out to be useful not only for providing a measure of the reliability of a distributed computer, but also for bounding the diameter costs of failures in multicomputer graphs and for determining an analytic lower bound on a general reliability measure, the Average Random Failure size.

4.2.1. Definitions

Given a graph $G = \{V, E\}$ that is at least 2-connected, it is always possible to reroute around an isolated failure. The cost of doing such rerouting locally will be measured by the minimum number of vertices

that must be involved in routing around a failure, and by the increase in the length of a path caused by routing within such a minimum set. Formally, for an edge $e = (v, w) \in E$, $\text{RegionSize}(e)$ is the number of vertices in the smallest cycle containing e . If all vertices within this cycle -- the rerouting region of e -- know that e has failed, no other vertex needs to change its routing algorithm: Any message being routed through e reaches, say, v normally, detours through the rerouting region to w , and then continues on its original route. Such a local rerouting increases the length of a path through e by at most $\text{RegionSize}(e) - 2$.

For a graph $G = \{V, E\}$, $\text{EdgeRegionSize}(G)$ is the maximum over all edges e in E of $\text{RegionSize}(e)$. An infinite family of graphs is edge reroutable if there is some fixed r such that $\text{EdgeRegionSize}(G) \leq r$ for every graph G in the family; EdgeRegionSize for the family is then the least such r .

The corresponding definition for the cost of locally rerouting around vertex-failures is slightly more complex. Given some vertex $v \in V$, $\text{RegionSize}(v)$ is $1 + |V_R(v)|$, where $V_R(v)$ is a minimal set of vertices satisfying

- a) $V_R(v)$ does not contain v ;
- b) given any pair of distinct neighbors of v , there is

a path between them using only vertices in $V_R(v)$. Any path through v may be detoured around v without the knowledge of vertices outside the rerouting region of v , $\{v\} \cup V_R(v)$. The $RerouteCost(v)$ of locally rerouting around v is the maximum over all pairs of neighbors of v of the difference between the length of the shortest path in $V_R(v)$ connecting them and their true distance.

For a graph $G = (V, E)$, $VertexRegionSize(G)$ is the maximum over all v in V of $RegionSize(v)$, and $RerouteCost(G)$ is the maximum over all v of $RerouteCost(v)$. In analogy with edge reroutability, an infinite family of graphs is vertex reroutable if there is some fixed r such that $VertexRegionSize(G) \leq r$ for every graph G in the family; $VertexRegionSize$ for the family is then the least such r , and $RerouteCost$ for the family is the maximum of $RerouteCost(G)$. A vertex reroutable family necessarily is edge reroutable and has a finite $RerouteCost$. Any vertex rerouting region must include all of the vertex's neighbors, so $VertexRegionSize$ for a graph is always at least $1 + d$.

To our knowledge, local reroutability is the only measure of its kind for comparing the suitability of infinite graph families for distributed computers. The importance of diameter has led to several extensions of connectivity to include diameter issues [59]. Hartman

and Rubin [60] have proposed the question of the diameter stability of graphs; Boesch [9] define the persistence of a graph as the smallest number of vertices whose removal increases the diameter. Perhaps the closest proposed criterion is given by Farley [61], who considers isolated failure immune graphs -- graphs that are not disconnected by any number of isolated (nonadjacent) failures. Unfortunately, the dense multicomputer graph families proposed are not immune to isolated failures; the families that are locally reroutable are, however, immune to failures separated by the diameter of the vertex rerouting region, which is typically small.

4.2.2. Reroutability of specific families

Although most members of the proposed multicomputer graph families are at least 2-connected, they vary considerably in their local rerouting costs -- both in the number of processors affected ($EdgeRegionSize$ and $VertexRegionSize$) and in the increased path lengths forced by local rerouting ($RerouteCost$). Indeed, several of the proposed infinite families are not vertex or even edge reroutable, despite being highly connected.

4.2.2.1. The hypercube

Theorem 2. The binary, complete, and mesh-connected hypercube families are edge reroutable, with $\text{EdgeRegionSize} \leq 4$, but are not vertex reroutable.

Proof. Each vertex of a graph in the graph product families can be assigned a (possibly mixed radix) address of n digits, where n is the number of graph operands, and the value of a particular digit denotes a vertex in the corresponding operand graph. An edge connects two addresses that differ in exactly one position, so each edge is in a 4-cycle whose edges respectively change a given digit, change any other digit, restore the given digit to its original value, and finally restore the other digit. A complete hypercube in which each operand graph has at least three vertices always has a 3-cycle for any edge (change a digit, change it to something else, restore it), and so has $\text{EdgeRegionSize} = 3$; other hypercubes have $\text{EdgeRegionSize} = 4$. Because the degrees of the members of each of these families increase without bound, the families are not vertex reroutable.

□

4.2.2.2. Tree variations

Theorem 3. The tree-based families of Friedman [22] are neither edge nor vertex reroutable.

Proof. In Friedman's construction, which joins d d -ary trees with L levels each by identifying their leaves, routing around a root or an edge adjacent to a root requires a path down through that tree, up to a root of another tree, and back again. Hence for a given graph, $\text{EdgeRegionSize} = 4L$, $\text{VertexRegionSize} = 2dL - d + 2$, and $\text{RerouteCost} = 4L - 4$.

□

Theorem 4. For each m , the family Hypertree m of Goodman and Sequin [62] is edge and vertex reroutable, with $\text{EdgeRegionSize} = 6$, $\text{VertexRegionSize} = 12 + 3m$, and $\text{RerouteCost} = 8$.

Proof. Goodman and Sequin present a general cycle for an edge in a Hypertree with four or more levels, which shows that the EdgeRegionSize is no more than 6. This cycle is shown in Figure 4.1a; it consists of two vertices joined by a hypercube-like edge, together with their parents and siblings. The distance up a tree from a level L to the nearest level whose hypercube-like edges complement the corresponding address bit is

$L/(1+m)$, so their cycle is minimal: A cycle whose length is independent of L and which includes one hypercube-like edge from level L must use another hypercube-like edge at the same level. Similarly, to connect a vertex to its grandparent without going through its parent, any vertex rerouting region must include a cycle at least as long as the 10-vertex cycle shown in Figure 4.1b. The vertex rerouting region is produced by merging these cycles, as shown in Figure 4.1c.

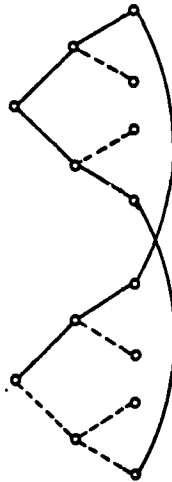
M

Figure 4.1

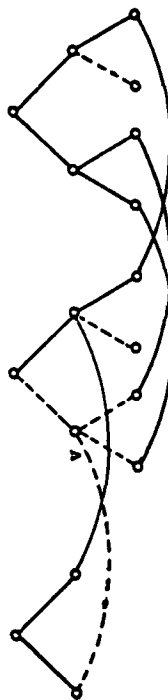
a) Edge rerouting region



b) Routing to the grandparent



c) Vertex rerouting region for v



Rerouting for the Hypertree

4.2.2.3. The cube-connected-cycles

Theorem 5. The cube-connected-cycles family (with $n > 1$) is edge and vertex reroutable, with $\text{EdgeRegionSize } 8$, $\text{VertexRegionSize } 14$, and $\text{RerouteCost } 10$.

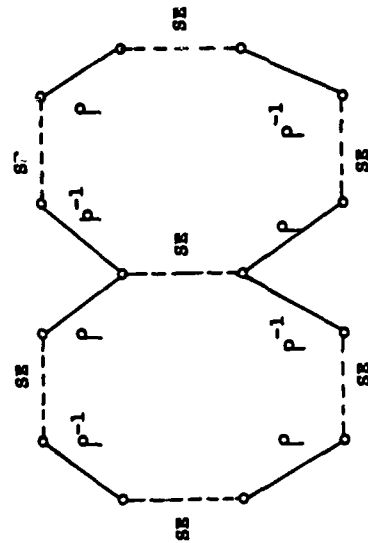
Proof. For a cube-connected-cycles graph with $n > 1$, an edge is always part of an 8-cycle of the form $p, SE, p^{-1}, SE, p, SE, p^{-1}, SE$, where p is an untwisted rotate; letting x denote the substring $x_1x_2 \dots x_{n-2}$, the addresses in our single-exchange formulation of the family are $\{s, x_0\bar{x}_{n-1}\}, \{s+1, \bar{x}_{n-1}x_0\}, \{s+1, \bar{x}_{n-1}\bar{x}_0\}, \{s, \bar{x}_0\bar{x}_{n-1}\}, \{s, \bar{x}_0x_{n-1}\}, \{s+1, \bar{x}_{n-1}\bar{x}_0\}, \{s+1, \bar{x}_{n-1}x_0\}$, and $\{s, x_0\bar{x}_{n-1}\}$. No two edges in this sequence are the same because no two addresses are identical: They disagree in either their stage components or corresponding bits of their binary string components.

No shorter cycle, or collection of shorter cycles, will contain every SE edge. Any cycle must have one SE between a p and a p^{-1} , and, by parity, an even number of SE operations. A cycle of length less than n must have exactly as many p as p^{-1} edges, in order to return to the original stage; however, between any two successive SE s, it must have an unequal number of p and p^{-1} edges, to avoid loops in the path. The only two sequences (ignoring cyclic permutations) of seven or fewer operations

that satisfy these constraints for n greater than 6 are $SE \ p \ SE \ p^{-1}$ and $SE \ p^2 \ SE \ p^{-2}$, neither of which defines a cycle. (For n less than 7, the constraints allow a few more sequences, none of which is a cycle.)

Combining two of these 8-cycles gives us the minimal vertex-rerouting set, with 14 vertices, as shown in Figure 4.2. All 14 addresses in this region are distinct, as before. Because the 8-cycles are minimal for rerouting around vertex v from $p^{-1}(v)$ or $p(v)$ to $SE(v)$, the set is minimal and VertexRegionSize is 14. If this region is unique, then the cube-connected-cycles have a RerouteCost of 10. Uniqueness follows from the uniqueness of our 8-cycle for edge rerouting, which is established by enumerating all sequences of eight operations satisfying the cycle constraints. Aside from cyclic permutations, the only such sequences are our edge cycle, $(SE \ p)^2 (SE \ p^{-1})^2$, $SE \ p^3 \ SE \ p^{-3}$, and a few additional sequences that only obey the constraints when n is less than 7. Of these, only our pattern is actually a cycle.

Figure 4.2



Vertex Rerouting for the Cube-Connected-Cycles

4.2.2.4. The shuffle-exchange

Theorem 6. The shuffle-exchange family is neither edge nor vertex reroutable.

Proof. Shuffle-exchange graphs are 1-connected.

W

Theorem 7. The elided shuffle-exchange family is edge and vertex reroutable, with $\text{EdgeRegionSize } 8$, $\text{VertexRegionSize } 14$, and $\text{RerouteCost } 10$.

Proof. For $n > 4$, an edge is always part of an 8-cycle with the pattern of Theorem 5: $p, SE, p^{-1}, SE, p, SE, p^{-1}, SE$, again letting x denote $x_1x_2 \dots x_{n-2}$, the corresponding addresses are $x_0x_{n-1}, x_{n-1}x_0, x_{n-1}x_0, x_{n-1}x_0, x_{n-1}x_0, x_{n-1}x_0, x_{n-1}x_0, x_{n-1}x_0$. Unlike Theorem 5, however, the addresses are not necessarily distinct and may represent elided vertices. Solving the possible equalities in the sequence shows that the only occasions where two nonadjacent addresses in the sequence can be equal are when the cycle contains two of the elided addresses, 0^* and 0^*1 , or their complements. Fortunately, the new edges created by the elisions each form a subsequence of the canonical 8-cycle, and so belong to smaller cycles: the edges from 10^* to 0^*10 and from 01^* to 1^*01 each represents the sequence $p, SE,$

p^{-1} , SE , and p (placing them in 4-cycles that end with SE, p^{-1}, SE), while for even n , the edge from $(01)^*00$ to $(10)^*11$ represents SE, p , and SE , placing it in a 6-cycle. Conversely, any canonical 8-cycle that contains an elided address must have it inside a subsequence represented by one of the new edges. For all n greater than 4, there are edges which do not belong to cycles containing new edges, so their $RegionSize$ s are actually 8. Minimality and uniqueness of the 8-cycle are proved by another enumeration of the possible short sequences; given these conditions, the vertex rerouting region argument follows the line of Theorem 5.

4.2.2.5. The lens

Theorem 8. A lens family of base b and degree $2b$ is edge and vertex reroutable, with $EdgeRegionSize$ 4, $VertexRegionSize$ $6+2b$, and $RerouteCost$ 4.

Proof. In the lens, $\{s, x\}$ is connected to $\{s+1, SE_1(p(x))\}$ for all $0 \leq i < b$, where p is the untwisted rotate. An edge always lies in a cycle of the form $\{s, x_0x\}, \{s+1, xi+x_0\}, \{s, \bar{x}_0x\}$, and $\{s+1, xi+\bar{x}_0\}$, where x is the substring $x_1 \dots x_{n-1}$ and $\bar{x}_0$ is any digit not equal to x_0 . These addresses are necessarily distinct, so the formal sequence does define an actual

cycle of length four. For n greater than 1, the edge $\{(0,0^*), \{1,0^*1\}\}$ is never in a 3-cycle, so $EdgeRegionSize$ is 4.

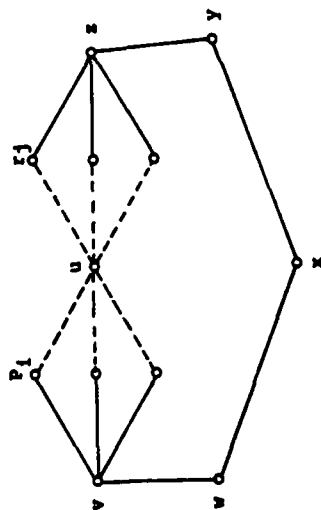
For n at least 6, a vertex $u = \{s, u_0u_1 \dots u_{n-1}\}$ has a rerouting region with $6+2b$ vertices. Letting $\bar{u}_0$ and $\bar{u}_{n-1}$ represent fixed b -ary digits not equal to u_0 and u_{n-1} respectively, and U denote the substring $u_1u_2 \dots u_{n-2}$, the region in general contains u , an outer path of 5 addresses ($v = \{s, u_0U\bar{u}_{n-1}\}$, $w = \{s+1, U\bar{u}_{n-1}\bar{u}_{n-1}\}$, $x = \{s, \bar{u}_0U\bar{u}_{n-1}\}$, $y = \{s-1, \bar{u}_0\bar{u}_0U\}$, and $z = \{s, \bar{u}_0U\bar{u}_{n-1}\}$), b addresses adjacent to both u and v ($P_i = \{s-1, iu_0U\} \forall 0 \leq i < b$), and b addresses adjacent to both u and z ($r_i = \{s+1, Uu_{n-1}i\} \forall 0 \leq i < b$). This set contains all neighbors of u , and, since no address in the outer path can equal u , it always provides a path from one neighbor to another without going through u . Figure 4.3 illustrates the region for a lens of degree 6.

This set is minimal, assuming that n is at least 6. First, we observe that the outer path is a shortest path between v and z . Any path between v and z must have either at least n edges or an even number of edges, because v and z are in the same stage. The current outer path has length 4, so any shorter path between them must have length 2; enumerating the neighbors of v and z

shows that they have none in common. (There are b^2 paths between v and z of length 4, all of which go through x ; the particular values of w and y were chosen so that the corresponding outer path can also be used for the de Bruijn proof below.) By a similar argument, the unique shortest path from any p_i to any r_j goes through v and z . Finally, enumerating the possible neighbors of the p_i shows that a rerouting region for u that does not contain an address of the form v would have at least $2+5b$ vertices just to provide a path between any two of the p_i ; similarly, a region without z could not have fewer than $2+5b$ vertices. So a minimal vertex set for rerouting around u must contain v and z ; to provide a path from the set of p_i to the set of r_j , it must either include the outer path (connecting v with z) or a path of length $\min(6, n-2)$ from some p_i to some r_j . Hence a lens with base b and n at least 6 has $\text{texRegionSize} = 6+2b$ and $\text{RerouteCost} = 4$. (For n less than 6, there is a shorter path between v and z that does not go through u : v connects to $\{s-1, p^{-1}(u)\}$, while z connects to $\{s+1, p(x)\}$. $\{s+1, p(x)\}$ and $\{s-1, p^{-1}(u)\}$ are neighbors of u , and are connected by $n-2$ edges, defining a vertex rerouting region of $2b+n$ vertices.)

W

Figure 4.3



Vertex Rerouting for the Lens

4.2.2.6. The de Bruijn and C'_g graphs

Corollary 8.1. A multistage de Bruijn of base b , degree $d = 2b$, and S at least 3 is edge and vertex reroutable, with $\text{EdgeRegionSize } 4$, $\text{VertexRegionSize } 6+d$, and $\text{RerouteCost } 4$.

Proof. Because S is at least 3, all the formal addresses used in the proof of Theorem 8 are necessarily distinct, so the proof carries over directly. W

Corollary 8.2. A C'_g graph of degree $d = 2m+1$, base $b = m(m+1)$, and S at least 4 is edge and vertex reroutable, with $\text{EdgeRegionSize } 4$, $\text{VertexRegionSize } 6+d$, and $\text{RerouteCost } 4$.

Proof. The edge patterns of Theorem 8 can be used directly; the only change in the vertex rerouting pattern required is that there are either m left neighbors P_1 and $m+1$ right neighbors, or vice versa, depending on whether the initial stage is odd or even.

W

Theorem 9. A de Bruijn family of base b and degree $2b$ is edge and vertex reroutable, with $\text{EdgeRegionSize } 4$, $\text{VertexRegionSize } 6+2b$, and $\text{RerouteCost } 4$.

Proof. The pattern of operations that worked for the lens works for the de Bruijn: every edge lies in a cycle of the form x_0x , x_1x_0 , $\bar{x}_0x$, and $x_1\bar{x}_0$. Unlike the cycle in the lens, these four addresses are not necessarily distinct; however an actual sequence of four such addresses can have at most two identical, and those two must be adjacent in the formal sequence: x cannot equal $\bar{x}_0x$ and x_1x_0 cannot equal $x_1\bar{x}_0$; if $x = x_1x_0$ then $x = a^n$ for some b -ary digit a , so $\bar{x}_0x = \bar{a}a^{n-1}$ while $x_1\bar{x}_0 = a^{n-1}\bar{a}$; an analogous situation holds if $\bar{x}_0x = x_1\bar{x}_0$; finally, x_1x_0 cannot equal $\bar{x}_0x$, and x cannot equal $x_1\bar{x}_0$. Hence the cycle of four formally distinct addresses must represent an actual cycle of either three or four distinct vertices. For $n > 2$, not all vertices in a de Bruijn are in 3-cycles so this edge rerouting region is minimal.

For $n > 5$, the minimal vertex rerouting region in a de Bruijn with base b has $6+2b$ formally distinct addresses, which are exactly analogous with those of the lens. The proofs that this vertex set provides paths between all neighbors of a vertex, and that no smaller set suffices for all addresses when n is greater than 5, are also analogous with those of Theorem 8, and are omitted. W

Theorem 10. Saluja and Reddy's 4-connected modification of the binary de Bruijn family is edge reroutable with $\text{EdgeRegionSize } 4$, but not vertex reroutable.

Proof. For given n , let Altern be $\{(01)^*, (10)^*\}$ for n even, and $\{(01)^*0, (10)^*1\}$ for n odd. Saluja and Reddy [58] make the binary de Bruijn network 4-connected by connecting each of 0^* and 1^* to both members of Altern. These new edges form a 4-cycle, and all other edges lie in 3-cycles or 4-cycles by Theorem 9, so the new family is edge reroutable. Consider, however, the vertex rerouting region for 0^* . The region must contain a path from the vertices 0^*1 and 10^* to the vertices in Altern that does not go through 0^* . If this path uses any of the new edges, it must go through 1^* .

Let p be a shortest path from any vertex in $\{0^*1, 10^*\}$ to any vertex in Altern $\cup \{1^*\}$ that does not go through 0^* . p must contain only edges in the unmodified de Bruijn; each such edge increases the number of ones in an address by at most one; therefore, p must contain at least $\lfloor n/2 \rfloor - 1$ edges to go from one of its initial addresses to any member of its destination addresses. Therefore, the vertex rerouting region for 0^* must contain more than $n/2$ vertices. (More detailed arguments show that the actual VertexRegionSize for one of these modified graphs is $n+2$ for n greater than 3).

4.2.2.7. The elided Moebius

Theorem 11. The elided Moebius family is edge and vertex reroutable, with $\text{EdgeRegionSize } 8$, $\text{VertexRegionSize } 14$, and $\text{RouteCost } 10$.

Proof. A sequence of address operations analogous to that for theorems 5 and 7 defines an 8-cycle for any particular edge in the Moebius graph: $p, DE, p^{-1}, DE, p, DE, p^{-1}, DE$, here p is the twisted rotate. Letting x represent the substring $x_1x_2 \dots x_{n-3}$, the addresses produced are $x = x_0x_{n-2}x_{n-1}, x_{n-2}x_{n-1}x_0, x_{n-2}x_{n-1}x_0, x_0x_{n-2}x_{n-1}, x_0x_{n-2}x_{n-1}, x_{n-2}x_{n-1}x_0, x_{n-2}x_{n-1}x_0, x_0x_{n-2}x_{n-1}$. Unlike Theorem 7, all these addresses are distinct: Because the twisted rotate changes a string's parity, the addresses fall into two sets of four having opposite parity, $\{x, x_0x_{n-2}x_{n-1}, x_0x_{n-2}x_{n-1}, x_0x_{n-2}x_{n-1}\}$ and $\{x_{n-2}x_{n-1}x_0, x_{n-2}x_{n-1}x_0, x_{n-2}x_{n-1}x_0, x_{n-2}x_{n-1}x_0\}$. Within the first set, the addresses are distinguished by their first and last bits; within the second, they are distinguished by their last two bits.

If n is odd, addresses in this cycle may represent elided vertices. An unelided address, however, could only be transformed to an elided one by a DE operation, since the two elided addresses are transformed into each

other by the ρ and ρ^{-1} operations. Thus the new edge created by elision represents both the sequence $DE \rho DE$ and the sequence $DE \rho^{-1} DE$; any formal cycle containing elided addresses then has a corresponding actual cycle in which one of these subsequences is replaced by a single edge. For n less than 6, there are shorter cycles for all edges, giving $EdgeRegionSizes$ of 3, 4, 6, and 7, respectively, for elided Moebius graphs with $n = 2, 3, 4$, and 5. For n at least 6, enumeration of the possible sequences of operations again shows that no other cycle or set of cycles with length less than 9 contains all edges; the description and proof of the vertex rerouting region therefore follow Theorem 5.

M

4.2.2.8. The elided even double-exchange

Theorem 12. The elided even double-exchange family is edge and vertex reroutable, with $EdgeRegionSize$ 8, $VertexRegionSize$ 14, and $RerouteCost$ 10.

Proof. The canonical 8-cycle, $\rho, DE, \rho^{-1}, DE, \rho, DE, \rho^{-1}, DE$ again provides the edge rerouting region for n at least 6; here ρ is untwisted rotation. The addresses produced are $x = x_0 \bar{x}_{n-2} \bar{x}_{n-1}, x_{n-2} \bar{x}_{n-1} \bar{x}_0, x_{n-2} \bar{x}_{n-1} \bar{x}_0, \bar{x}_0 \bar{x}_{n-2} \bar{x}_{n-1}, \bar{x}_0 \bar{x}_{n-2} \bar{x}_{n-1}, \bar{x}_{n-2} \bar{x}_{n-1} \bar{x}_0, \bar{x}_{n-2} \bar{x}_{n-1} \bar{x}_0$, and $x_0 \bar{x}_{n-2} \bar{x}_{n-1}$. Unlike Theorem 11, the ρ operation does

not alter parity, and not all these addresses are necessarily distinct in the nonelided version of the family. However, the elisions remove precisely those cases where there are duplicated addresses: any address that returned to itself after a sequence of one or two operations would have degree less than 3, and be elided; the only remaining possible looping subsequences are $DE \rho DE$ and $DE \rho^{-1} DE$, both of which imply that any starting address that returns to itself has a DE edge to 0^* or 1^* , and so is elided. Conversely, each of the new edges added by elision represents a subsequence in this cycle, and so belongs to a cycle of length less than 8.

There are no elided even double-exchange graphs for n less than 5. When n is 5, $EdgeRegionSize$ is 7; for larger values of n , minimality and uniqueness of the 8-cycle is established by enumerating the possible operation sequences, so the vertex rerouting arguments of Theorem 5 may be applied again.

M

4.2.2.9. Summary

Table 4.3 summarizes the $RegionSizes$ and $RerouteCosts$ for local reroutability in the proposed multicomputer graph families. In particular, the table shows that the elided even double-exchange and Moebius

families are more reliable than the densest previous trivalent family, the shuffle-exchange, and as reliable as the even less dense cube-connected-cycles.

Table 4.3
Region Sizes and Reroute Costs for Graph Families

	Edge Region Size	Vertex Region Size	Reroute Cost
Graph product families	4	-	-
Tree families			
trees	-	-	-
Friedman's trees	-	-	-
Hypertree m	6	$12+3m$	8
Single-exchange families			
cube-connected-cycles	8	14	10
shuffle-exchange	-	-	-
elided shuffle-exchange	8	14	10
lens, degree $d = 2b$	4	$6+d$	4
multistage de Bruijn	4	$6+d$	4
C_8^3 Bruijn	4	$6+d$	4
d_8^3 Bruijn	4	$6+d$	4
Saluja and Reddy	4	-	-
Double-exchange families			
elided Moebius	8	14	10
elided even	8	14	10
double-exchange	8	14	10

4.3. Average Random Failsets

Under the worst possible set of simultaneous failures, a multicomputer graph of connectivity c can survive $c-1$ vertex failures, but not c failures. This worst case, however, is unlikely: An elided Moebius graph is disconnected by three failures only if they are the three neighbors of some one vertex, or if, among other constraints, they lie within distance 1 of a 3-cycle. Regardless of N , an elided Moebius graph never has more than four 3-cycles, so if the three failures are uniformly randomly distributed the probability of their disconnecting the graph is $6/(N-1)(N-2) + O(N^{-3})$. On the other extreme, the local rerouting properties of Moebius graphs show that in the best possible arrangement of failures, where all failures are at least distance 4 apart, there is no limit to the number of vertex or edge failures that can be sustained (see Lemma 13.1 below). Neither extreme in the distribution of failures is likely; a more representative measure of reliability would be the expected number of random failures required to disconnect a graph. This number is called the Average Random Failset size (ARF size) of the graph.

4.3.1. Definitions and basic results

What, exactly, does "the expected number of failures required to disconnect a graph" mean? In analogy with connectivity, the failures may be either vertices or edges; an edge-ARF concerns failed edges while a vertex-ARF concerns vertex failures. Our model assumes that the failures are uniformly randomly distributed among the nonfailed elements (vertices or edges), without replacement. A graph's ARF size is then $\sum p(i)$, where $0 < i$ and $p(i)$ is the probability that exactly i failures are required to disconnect the graph (equivalently, that the i -th random failure disconnects the graph, while $i-1$ failures do not). Since a failure is held to disconnect a graph if there are two vertices in the resulting graph with no path between them, or if the resulting graph has less than two vertices (the trivial graph), the summation only goes through $i = N-1$ for vertex-ARFs and $i = |E| - (N-2)$ for edge-ARFs.

The great complexity of the probabilities $p(i)$ for nontrivial graph families has led us to three alternative approaches to measuring their ARF sizes. For a few very simple families, we can derive closed-form approximations for the sum. For more complex families, we can determine the ARF sizes for graphs with no more than a few thousand vertices by simulation; the close

correspondence between the results of the approximations and the simulations for the simple graph families provides some confidence in our implementation. A more general approach will be presented in the last section, where we derive a closed formula that is a lower bound for the ARP size of any reroutable family, in terms of its edge or vertex RegionSize.

The ARP size formula can be rewritten as $\sum q(i)$, where $q(i)$ is the probability that at least i failures are required to disconnect the graph (equivalently, that $i-1$ failures do not disconnect it). For sufficiently simple families of graphs, the $q(i)$ may be derived immediately. A line of N processors is disconnected if any edge fails, so its edge-ARP size is exactly 1. A single vertex-failure does not, however, disconnect a line if N is greater than 2 and the failed vertex is at one end of the line. For the vertex-ARP, therefore, $q(1)$ is 1 and $q(i)$ is $q(i-1) \cdot 2/(N+2-i)$. The vertex-ARP size is then $\sum_{i=1}^{N+1} 1/(N+1-i)!$, or $1 + 2/N + O(N^{-2})$. Similarly, a ring of processors, C_N , has an edge-ARP size of 2 and a vertex-ARP size of $2 + 2/(N-1) + O(N^{-2})$. All trees have an edge-ARP size of 1; for an m -ary tree the vertex-ARP size is nearly m . A vertex failure disconnects the tree if and only if the vertex is not a leaf. An m -ary tree with L levels below the root has m^L leaves

out of $m^{L+1}-1/(m-1)$ vertices; when the number of vertices is large, $(m-1)/m$ of them are leaves, and $q(i) = ((m-1)/m)^{i-1}$, so the ARP size is approximately m .

These simple results are useful because they validate the results of our simulations, which can then be used to determine the ARP sizes for graphs with more complex structure. Table 4.4 lists the graphs simulated and the ARP size found for 1000 iterations on each graph. Since each mean represents 1000 regeneration cycles, the confidence interval listed is calculated on the assumption that the means are normally distributed random variables. We have an exact summation formula for the vertex-ARP size of a line of processors, so the true ARP sizes are given in parentheses after the simulation results for lines; of the fifteen values of N tested, all of the true answers lie within the 95% confidence interval of the corresponding simulation.

Table 4.4

ARP Simulation Results

N	simulation value	exact value	confidence interval
---	---------------------	----------------	------------------------

Degree 2

lines

2	1.000	(1.000)	0.00
3	1.665	(1.667)	0.029
4	1.836	(1.833)	0.055
5	1.755	(1.733)	0.067
6	1.540	(1.578)	0.059
7	1.450	(1.451)	0.057
8	1.335	(1.363)	0.045
9	1.275	(1.303)	0.037
10	1.236	(1.261)	0.035
12	1.238	(1.205)	0.037
15	1.169	(1.156)	0.027
20	1.098	(1.112)	0.021
40	1.048	(1.053)	0.014
100	1.016	(1.020)	0.008

Table 4.4 (continued)

N	simulation value	confidence interval
---	---------------------	------------------------

Degree 3

trees

7	2.031	0.080
15	2.037	0.083
31	1.994	0.082
63	2.057	0.091
127	1.976	0.084
255	1.968	0.084
511	2.069	0.092
1023	1.998	0.088
2047	1.994	0.084
4095	1.978	0.095

cube-connected-cycles

8	2.43	0.05
24	6.12	0.09
64	13.82	0.24
160	26.61	0.54
384	47.20	0.96
896	82.63	1.77
2048	142.42	3.01

Table 4.4 (continued)

N	simulation value	confidence interval
Degree 3		
elided shuffle-exchange		
4	1.86	0.06
8	2.61	0.10
16	3.61	0.12
32	6.71	0.19
64	10.18	0.30
128	18.24	0.54
256	29.05	0.85
512	49.41	1.36
1024	79.24	2.07
2048	132.36	3.42
4096	210.71	5.35
elided even double-exchange		
2	1.00	0.00
4	2.33	0.05
8	3.54	0.09
16	4.52	0.13
32	7.43	0.16
64	11.79	0.30
128	21.37	0.44
256	31.96	0.78
512	55.26	1.20
1024	83.17	2.10
2048	141.31	3.09
elided Moebius		
4	3.00	0.00
8	3.89	0.09
16	5.57	0.09
32	8.30	0.16
64	13.61	0.25
128	21.46	0.44
256	34.57	0.71
512	54.98	1.22
1024	88.75	2.02
2048	142.68	3.12
4096	225.86	5.18

Table 4.4 (continued)

N	simulation value	confidence interval
Degree 4		
lens		
8	4.933	0.086
24	10.714	0.151
64	22.10	0.34
160	43.27	0.73
384	81.60	1.43
896	153.26	2.51
2048	278.63	15.15
de Bruijn		
4	2.84	0.02
8	4.74	0.10
16	6.98	0.15
32	11.42	0.21
64	19.84	0.36
128	33.79	0.65
256	57.56	1.10
512	97.43	1.79
1024	163.72	3.06
2048	278.81	5.13
tree		
13	2.850	0.124
40	2.933	0.140
121	2.941	0.150
364	2.995	0.151
1093	3.112	0.156
3280	3.057	0.161
hypertree 1		
15	4.944	0.138
31	6.620	0.184
63	9.573	0.28
127	13.51	0.39
255	18.45	0.57
511	26.64	0.81
1023	36.84	1.14

Table 4.4 (continued)

M simulation value confidence interval

Degree 5

tree	21	3.792	0.171
	85	3.823	0.195
	341	3.996	0.20
	1365	3.942	0.21
	5461	4.101	0.22

hypertree 2

15	6.809	0.143
31	10.632	0.20
63	17.27	0.35
127	27.76	0.58
255	43.63	0.95
511	69.96	1.56

Unbounded Degreebinary hypercube

2	1.00	0.00
4	2.67	0.03
8	4.93	0.08
16	8.86	0.13
32	16.98	0.21
64	33.71	0.39
128	66.79	0.71
256	132.43	1.23
512	263.77	2.26
1024	520.62	4.11

4.3.2. An analytic lower bound

The simulation results are instructive for moderately large graphs, but fail to supply provable bounds on the ARP sizes for the interesting multicomputer graph families. Fortunately, a general formula for a lower bound of the ARP sizes for most multicomputer graph families can be derived from the ability to reroute around a single failure using purely local knowledge.

Theorem 13. A graph with $\text{VertexRegionSize } R$ has a vertex-ARP size of at least $(nR/2R)^{1/2} - 1/3$.

We will derive this lower bound by defining a restricted class of sequences of failures which do not disconnect their graph (Lemma 13.1) and whose probabilities are easily bounded.

Lemma 13.1. Given a 2-connected graph G , and vertices $v_1, \dots, v_m$ such that v_i is not in $V_R(v_j) \forall j < i$, $G - \{v_1, \dots, v_m\}$ is connected.

Proof. By induction on m : If m is 1, $G - \{v_1\}$ is connected by the definition of 2-connectedness. Assume that the graph $G - \{v_1, \dots, v_{m-1}\}$ is connected, so, in particular, for any two vertices u and $w \in V - \{v_1, \dots, v_{m-1}\}$ there is a path P_u in G , $u = w_1 \dots w_p = w$, such that v

$1 \leq i \leq p$, $w_i \in V - \{v_1, \dots, v_{m-1}\}$. $G - \{v_1, \dots, v_m\}$ is connected if there is a path between u and w all of whose vertices are in $V - \{v_1, \dots, v_m\}$. If v_m does not appear in P_u , P_w is the required path between u and w . Otherwise, suppose that $w_i = v_m$, w_{i-1} and w_{i+1} cannot be in $V - \{v_1, \dots, v_m\}$ because v_m cannot be a neighbor of any v_j for $j < m$. There is a path P_x in G , $w_{i-1} = x_1 \dots x_q = w_{i+1}$, all of whose vertices lie in $V_R(v_m)$. If this path does not contain any vertices in $\{v_1, \dots, v_{m-1}\}$, we can produce the required path between u and w by using P_x to route between w_{i-1} to w_{i+1} in P_w . If P_x does contain some forbidden vertices, we remove them from the path one at a time: Let x_r in P_x be the vertex v_j that has the highest subscript j in the sequence $v_1, \dots, v_m$. There is a path P_y in G , $x_{r-1} = y_1 \dots y_s = x_{r+1}$ that lies entirely in $V_R(x_r)$, and therefore lies in $G - \{v_1, \dots, v_{j-1}\}$. We modify P_x by replacing $x_{r-1} x_r x_{r+1}$ with P_y and erasing any resulting loops; if the resulting path has any forbidden vertices, their indices in $\{v_1, \dots, v_m\}$ must be less than j . By repeating this operation at most $j-1$ times on the successively modified paths P_x , we produce a path P_x in $G - \{v_1, \dots, v_m\}$ that takes w_{i-1} to w_{i+1} . The original path P_w can now be modified to avoid v_m (and all other forbidden vertices) by replacing $w_{i-1} v_m w_{i+1}$ with the final P_x and erasing

any resulting loops. Therefore there is a path between any vertices in $V - \{v_1, \dots, v_m\}$ that lies entirely in $V - \{v_1, \dots, v_m\}$, so $G - \{v_1, \dots, v_m\}$ is connected, completing the proof of Lemma 13.1. M

Proof of Theorem 13.

The vertex-ARF size formula can be written as $\Sigma q(m)$, where $1 \leq m < N$, and $q(m)$ is the probability that $m-1$ vertex failures do not disconnect the graph. A lower bound for each $q(m)$ therefore provides a lower bound for the vertex-ARF size. The graph G is connected, so $q(1) = 1$. If the $m-1$ vertex failures, $v_1, \dots, v_{m-1}$, satisfy the condition of Lemma 13.1, the graph is connected. Enumerating such sets, we see that v_i can be any one of the $N - |\bigcup V_R(v_j)| \leq N - (i-1)R$ vertices not belonging to the rerouting regions of vertices v_j for $j < i$. Letting $\bar{N}$ denote N/R , then for i greater than 1

$$q(i) \geq q(i-1) (N - (i-1)R)/N \geq q(i-1) (\bar{N} - (i-1))/\bar{N}.$$

Solving this last recurrence shows that

$$q(i) \geq \bar{N} / (\bar{N}^i (\bar{N} - i)),$$

so the vertex-ARF size is greater than or equal to

$$\sum_{1 \leq i \leq \bar{N}} \bar{N} / (\bar{N}^i (\bar{N} - i)).$$

This formula can be expressed in terms of the function

denoted $Q(m)$ in Knuth [63, p. 112],

$$Q(m) = \sum_{1 \leq i \leq m} m!/(m-i)! m^i,$$

showing that the vertex-ARF size for a graph with $\text{VertexRegionSize } R$ is greater than $Q(\tilde{R})$. $Q(m)$, however, has a well-known series expansion [63, p. 117]:

$$Q(m) = (m/2)^{1/2} - 1/3 + O(m^{-1/2}),$$

where the $O(m^{-1/2})$ term is always positive. Therefore the vertex-ARF size for a graph with $\text{VertexRegionSize } R$ is at least $(m/2R)^{1/2} - 1/3$, completing the proof of Theorem 13.

M

Theorem 14. A graph with $\text{EdgeRegionSize } R$ has an edge-ARF size of at least $(m/2R)^{1/2} - 1/3$.

Proof. The proof exactly parallels the proof for Theorem 13.

M

The region sizes listed in Table 4.3 can now be used to derive lower bounds for the ARF sizes of all the reroutable families studied; Table 4.5 compares these bounds with the ARF sizes derived by simulation for moderately large graphs in these families.

Table 4.5
Comparison of ARF Simulation and Analytic Bound

Degree $\bar{d}$	N	simulation result	analytic bound
cube-connected-cycles			
160		26.61	3.91
384		48.20	6.23
896		82.63	9.70
2048		142.42	14.83
elided shuffle-exchange			
256		29.05	5.03
512		49.41	7.25
1024		79.24	10.39
2048		132.36	14.83
4096		210.71	21.11
elided even double-exchange			
256		31.96	5.03
512		55.26	7.25
1024		83.17	10.39
2048		141.31	14.83
elided Moebius			
256		34.57	5.03
512		54.98	7.25
1024		88.75	10.39
2048		142.68	14.83
4096		225.86	21.11

AD-A123 586

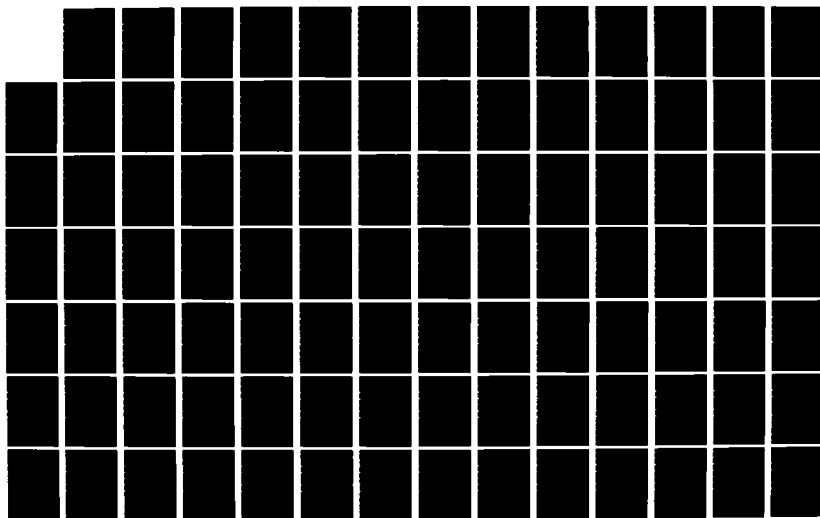
OPERATING SYSTEMS FOR RING-BASED MULTIPROCESSORS(U)
WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES
R A FINKEL ET AL. 1982 N00014-81-C-2151

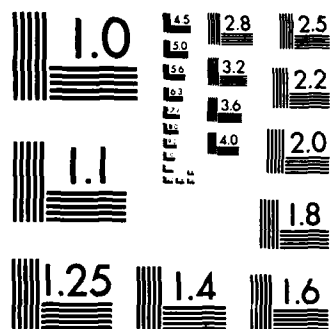
3/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Table 4.5 (continued)

N	simulation result	analytic bound
<u>Degree 4</u>		
lens		
160	43.27	4.68
384	81.60	7.44
896	153.26	11.53
2048	278.83	17.61
de Bruijn		
256	57.56	6.01
512	97.43	8.64
1024	163.72	12.34
2048	278.81	17.61
hypertree 1		
255	18.45	4.84
511	26.64	6.99
1023	36.84	10.02
<u>Degree 5</u>		
hypertree 2		
255	43.63	4.39
511	69.96	6.35

4.4. Summary

This chapter has compared the reliability of many proposed multicomputer graphs. After showing that the even double-exchange and Moebius families can be easily modified to satisfy the conventional graph-theoretic reliability criterion of connectivity, we introduced two additional criteria: the ease of locally rerouting around failures, and the expected number of uniform random failures required to disconnect a graph (its ARF size). We have shown that these two criteria are closely related -- one measure of local (edge or vertex) reroutability, the RegionSize, provides a lower bound for the corresponding ARF size. Both the traditional standard of connectivity and our proposed reliability measures show that the elided even double-exchange and Moebius families are highly reliable, in addition to being denser than any previously proposed trivalent multicomputer graph.

Chapter 5

Conclusions

5.1. Summary

We have studied two important classes of criteria for effective multicomputer interconnection topologies, relating to density and reliability. Chapter 2 compared the known topologies in terms of density, interpolability, and routing algorithms. Here we saw that the densest families -- those with the least message transit times -- had diameters logarithmic in the number of vertices. Most of these highly dense families fell within the definitional framework of the single-exchange construction, which provided a uniform diameter bound, linear interpolability, and a simple routing algorithm. For the small degrees that we studied, still denser families were constructed using the related double-exchange operation. This method of attaining better density still allowed interpolability and simple routing; in the specific case of the even double-exchange graphs, the practical utility of the new family was further supported in Chapter 3 by demonstrating a computationally uniform emulation of the well-known shuffle-exchange network and a compact VLSI layout.

Chapter 4 began our investigation of the reliability of dense multicomputer graphs by summarizing the connectivities of known families. Eliding vertices with low degree from the densest trivalent graphs, the shuffle-exchange and two denser double-exchange families, made them 3-connected without harming their diameter, degree, routing, interpolability, or layout properties. The elided families were also shown to satisfy the additional reliability criterion of local reroutability, unlike other proposed modifications in which edges are added to existing vertices. Our analysis of local reroutability provided two formal measures of the cost of handling a failure by transparently detouring messages around it: the RegionSize and the RerouteCost. After deriving these costs for the dense multicomputer graph families, Chapter 4 then proved that they also provided a lower bound for our final reliability criterion, Average Random Failure size.

5.2. Future Research

Despite the progress made in designing high density graph families with small degrees, much further work remains to be done. The Moore bound still provides the only known lower bound on $k/\lg N$ for infinite families; raising this bound will surely require more powerful use of graph theory. Pragmatically, the bound suggests that there may be far denser multicomputer graph families still to be found; at degree 3, for example, we have lowered the attainable $k/\lg N$ from 2 to 1.47, but the only known limit is 1.

Linear interpolability gives the designer of multicomputers great freedom in choosing the numbers of processors used, but does not adequately answer the problem of extensibility: can processors be added cheaply to an existing network? Families with fixed degree at least guarantee that the processors will not need to be radically modified, but designs should demand only small amounts of line and routing changes for small additions. The tree variations clearly satisfy this criterion, but we have not yet found any attack on the question for the single-exchange and double-exchange families.

We seem to be close to finding an asymptotically optimal layout for the even double-exchange, using our homomorphism and the known optimal layouts for the

shuffle-exchange; if a similar homomorphism could be found for the Moebius, that family would be still more appealing.

Local reroutability not only provides a suitable measure of the cost of distributed failure recovery, but also provides a tractable lower bound for the Average Random Failset size of many desirable topologies. We expect to finish deriving local rerouting costs for other dense double-exchange families shortly. For the longer term, we are interested in the effects of multiple simultaneous failures. Local reroutability already guarantees that multiple failures can be handled when they are sufficiently far apart; if we can provide limits on the rerouting regions needed for closer failures, these may in turn allow tighter bounds to be derived for ARP sizes. In any case, ad hoc arguments may provide much more precise ARP sizes for specific families.

The usefulness of an implementation depends on many other criteria. For fault tolerance in particular, a network needs to be able to diagnose failures in a distributed fashion. Ideally, a diagnosability model could be designed for the general single-exchange and double-exchange constructions. Another critical issue is traffic congestion; algorithms that communicate only with nearby vertices may therefore be important, and may

make the conventional cost criteria of diameter and average distance less applicable. Our homomorphism guarantees that algorithms with good locality in the shuffle-exchange would also be local in the double-exchange; this approach needs to be extended to interrelated more multicomputer graph families.

5.3. Conclusions

Multicomputer interconnection topologies with small degree and high density are essential to achieving in practice the increased processing power promised by recent advances in microprocessor and VLSI technology. Our double-exchange families offer superior density for small degrees, without sacrificing other practical requirements, involving routing and reliability.

References

- [1] D. H. Lawrie, "Access and alignment of data in an array processor," IEEE Transactions on Computers C-24, 12, pp. 1145-1155 (December 1975).
- [2] T. Lang, "Interconnections between processors and memory modules using the shuffle-exchange network," IEEE Transactions on Computers C-25, 5, pp. 496-503 (May 1976).
- [3] D. Hoey and C. E. Leiserson, "A layout for the shuffle-exchange network," Proc. 1980 International Conference on Parallel Processing, pp. 329-336 (August 1980).
- [4] D. Kleitman, P. T. Leighton, M. Lepley, and G. L. Miller, "New layouts for the shuffle-exchange graph," Proc. 13th Annual ACM Symposium on the Theory of Computing, pp. 278-292 (11-13 May 1981).
- [5] F. Harary, Graph Theory, Addison-Wesley, Reading, Massachusetts (1969).
- [6] E. Chang, "An introduction to echo algorithms," Proc. 1st International Conference on Distributed Computers, pp. 193-198 (October 1979).
- [7] R. S. Wilkov, "Analysis and design of reliable computer networks," IEEE Transactions on Communications COM-20, 3, pp. 660-678 (June 1972).
- [8] Y. J. Park and S. Tanaka, "Reliability evaluation of a network with delay," IEEE Transactions on Reliability R-28, 4, pp. 320-324 (October 1979).
- [9] F. T. Boesch, F. Harary, and J. A. Kabell, "Graphs as models of communication network vulnerability: connectivity and persistence," Networks 11, 1, pp. 57-63 (Spring 1981).
- [10] A. J. Hoffman and R. R. Singleton, "On Moore graphs with diameters 2 and 3," IBM Journal of Research and Development 4, 5, pp. 497-504 (November 1960).
- [11] E. Bannai and T. Ito, "On finite Moore graphs," Journal of the Faculty of Science of the University of Tokyo, Section 1A 20, 2, pp. 191-208 (August 1973).
- [12] R. M. Damerell, "On Moore graphs," Proc. Cambridge Philosophical Society 74, 2, pp. 227-236 (September 1973).
- [13] P. Erdos, S. Fajtlowicz, and A. J. Hoffman, "Maximum degree in graphs of diameter 2," Networks 10, 1, pp. 87-90 (Spring 1980).
- [14] J. R. Goodman and A. M. Despain, "A study of the interconnection of multiple processors in a data base environment," Proc. 1980 International Conference on Parallel Processing, pp. 269-278 (August 1980).
- [15] B. W. Arden and H. Lee, "Analysis of chordal ring network," Proc. Workshop on Interconnection Networks for Parallel and Distributed Processing, pp. 93-100 (April 1980).
- [16] C. H. Sequin, "Doubly twisted torus networks for VLSI processor arrays," Proc. 8th Symposium on Computer Architecture, pp. 471-480 (May 1981).
- [17] R. A. Finkel and M. H. Solomon, "Processor interconnection strategies," IEEE Transactions on Computers C-29, 5, pp. 360-371 (May 1980).
- [18] R. Sullivan, T. R. Bashkov, and K. Klappholz, "A large scale, homogeneous, fully distributed parallel machine," Proc. 4th Symposium on Computer Architecture, pp. 105-124 (March 1977).
- [19] J. C. Bermond and B. Bollobas, "The diameter of graphs - a survey," Proceedings of the 12th Southeastern Conference on Combinatorics, Graph Theory, and Computing, (December 1981).
- [20] L. N. Bhuyan and D. P. Agrawal, "A general class of processor interconnection strategies," Proc. 9th Symposium on Computer Architecture, pp. 90-98 (April 1982).

- [21] L. D. Wittie, "Communication structures for large networks of microcomputers," IEEE Transactions on Computers C-30, 4, pp. 264-273 (April 1981).
- [22] H. D. Friedman, "A design for (d,k) graphs," IEEE Transactions on Electronic Computers EC-15, 4, pp. 253-254 (April 1966).
- [23] B. E. Arden and H. Lee, "A multi-tree-structured network," Proceedings of Compcon, Computer Communications Networks, pp. 201-210 (Fall 1978).
- [24] J. R. Goodman and C. H. Sequin, "Hypertree, a multiprocessor interconnection topology," IEEE Transactions on Computers, (1979) Submitted for publication.
- [25] S. W. Golomb, "Permutations by cutting and shuffling," SIAM Review 3, pp. 293-297 (October 1961).
- [26] H. S. Stone, "Parallel processing with the perfect shuffle," IEEE Transactions on Computers C-20, 2, pp. 153-161 (February 1971).
- [27] F. P. Preparata and J. Vuillemin, "The cube-connected-cycles: A versatile network for parallel computation," CACM 24, 5, pp. 300-309 (May 1981).
- [28] R. A. Finkel and M. H. Solomon, "The Lens Interprocessor Connection Strategy," IEEE Transactions on Computers C-30, 12, pp. 960-965 (December 1981).
- [29] G. Farhi, "Diametres dans les graphes et rotations gracieuses," Ph.D. Thesis, l'Universite de Paris Sud (30 April 1981).
- [30] D. S. Parker, Jr., "Notes on shuffle/exchange-type switching networks," IEEE Transactions on Computers C-29, 3, pp. 213-222 (March 1980).
- [31] H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," Proc. 5th Annual Symposium on Computer Architecture, pp. 273-279 (April 1978).
- [32] T. Lang and H. S. Stone, "A shuffle-exchange network with simplified control," IEEE Transactions on Computers C-25, 1, pp. 55-65 (January 1976).

- [33] L. R. Goke and G. J. Lipowski, "Banyan networks for partitioning multiprocessor systems," 1st Annual Symposium on Computer Architecture, pp. 21-28 (December 1973).
- [34] M. C. Pease, "The indirect binary n-cube microprocessor array," IEEE Transactions on Computers C-26, 5, pp. 458-473 (May 1977).
- [35] D. G. de Bruijn, "A combinatorial problem," Koninklijke Nederlandse Academie van Wetenschappen te Amsterdam, Proc. Section of Sciences 49, 7, pp. 758-764 (1946).
- [36] H. S. Stone, "Dynamic memories with enhanced data access," IEEE Transactions on Computers C-21, 4, pp. 359-366 (April 1972).
- [37] Y. V. Golunkov, "Automation program realization of substitutions of symmetric semigroups II," Kibernetika 5, pp. 35-42 (1975).
- [38] C. W. H. Lam, "On some solutions of $A_k = dI + \lambda \text{Id}$," Journal of Combinatorial Theory, Series A 23, pp. 140-147 (1977).
- [39] M. Imase and M. Itoh, "Design to minimize diameter on building-block network," IEEE Transactions on Computers C-30, 6, pp. 439-442 (June 1981).
- [40] D. K. Pradhan, "Interconnection topologies for fault-tolerant parallel and distributed architectures," Proc. 1981 International Conference on Parallel Processing, pp. 238-244 (25 August 1981).
- [41] R. M. Storwick, "Improved construction techniques for (d,k) graphs," IEEE Transactions on Computers C-19, 12, pp. 1214-1216 (December 1970).
- [42] W. E. Leland, R. A. Finkel, Li Qiao, M. H. Solomon, and L. Uhr, "High density graphs for processor interconnection," Information Processing Letters 12, 3, pp. 117-120 (13 June 1981).
- [43] J. C. Bermond, C. Delorme, and G. Farhi, "Large graphs with given degree and diameter II," Rapport de recherche 105, Universite de Paris-sud, Centre d'Orsay, Laboratoire de Recherche en Informatique (June 1981).

- [44] J. C. Bermond, C. Delorme, and G. Fathi, "Large graphs with given degree and diameter III," Rapport de recherche 104, Université de Paris-sud, Centre d'Orsay, Laboratoire de Recherche en Informatique (September 1981).
- [45] J. J. Quisquater, "New constructions of large graphs with fixed degree and diameter, To appear."
- [46] J. C. Bermond, C. Delorme, and J. J. Quisquater, "Grands graphes non dirigés de degré et diamètre fixes," Rapport de recherche 113, Université de Paris-sud, Centre d'Orsay, Laboratoire de Recherche en Informatique (December 1981).
- [47] W. E. Leland and M. H. Solomon, "Dense trivalent graphs for processor interconnection," IEEE Transactions on Computers C-31, 3, pp. 219-223 (March 1982).
- [48] M. C. Pease, "An adaptation of the Fast Fourier Transform for parallel processing," Journal of the ACM 15, 2, pp. 252-264 (April 1968).
- [49] J. T. Schwartz, "Ultracomputers," ACM Transactions on Programming Languages and Systems 2, 4, pp. 484-521 (October 1980).
- [50] J. P. Flaminio and R. A. Finkel, "Quotient networks," IEEE Transactions on Computers C-31, 4, (April 1981).
- [51] C. D. Thompson, "Area-time complexity for VLSI," Proc. 11th Annual ACM Symposium on the Theory of Computing, pp. 81-86 (May 1979).
- [52] C. D. Thompson, "A Complexity Theory for VLSI," Ph.D. Thesis, Carnegie-Mellon University Computer Science Department (September, 1979).
- [53] H. Frank and I. T. Frisch, "Analysis and design of survivable networks," IEEE Transactions on Communications Technology COM-18, pp. 501-519 (October 1970).
- [54] I. Rubin, "On reliable topological structures for message-switching communication networks," IEEE Transactions on Communications COM-26, 1, pp. 62-74 (January 1978).

- [55] D. Minoli and E. H. Lipper, "Cost implications of survivability of terrestrial networks under malicious failure," IEEE Transactions on Communications COM-28, 9, pp. 1668-1674 (September 1980).
- [56] K. Steiglitz, P. Wiener, and D. J. Kleitman, "The design of minimum-cost survivable networks," IEEE Transactions on Circuit Theory 16, pp. 455-460 (November 1969).
- [57] F. T. Boesch and R. E. Thomas, "On graphs of invulnerable communication networks," IEEE Transactions on Communications Technology COM-18, pp. 484-489 (May 1970).
- [58] K. K. Saluja and S. M. Reddy, "A class of undirected graphs," Proc. 15th Annual Conference on Information Systems and Sciences, pp. 388-393 (March 1981).
- [59] B. Bollobas, "A problem of the theory of communication networks," pp. 29-36 in Theory of Graphs, Akad. Kaïdo, Budapest (1968) edited by G. Katona and P. Erdos.
- [60] J. Hartman and I. Rubin, "On diameter stability of graphs," pp. 247-254 in Theory and Applications of Graphs, Springer-Verlag; Lecture Notes 642 (1978) edited by Y. Alavi and D. R. Lick.
- [61] A. M. Farley, "Networks immune to isolated failures," Networks 11, 3, pp. 255-268 (Fall 1981).
- [62] J. R. Goodman and C. H. Sequin, "Hypertree, a multiprocessor interconnection topology," Technical Report 427, University of Wisconsin--Madison Computer Sciences (April 1981).
- [63] D. E. Knuth, The Art of Computer Programming Volume 1--Fundamental Algorithms (second edition), Addison-Wesley (1973).

APPENDIX C

**** High Density Graphs for Processor Interconnection**

*Will Leland
Raphael Finkel
Li Qiao
Marvin Solomon
Leonard Uhr*

Information Processing Letters
Vol. 12, No. 3 (13 June 1981)

**This material was published in Information Processing Letters, Vol. 12
No. 3 (13 June 1981), pp. 117-120, North Holland Publishing Co., Amsterdam."

APPENDIX D

A Stable Distributed Scheduling Algorithm

Raymond M. Bryant

Raphael A. Finkel

Proceedings of the Second International Conference on
Computing Systems, Paris France, April 1981

A STABLE DISTRIBUTED SCHEDULING ALGORITHM

Raymond M. Bryant
and
Raphael. A. Finkel

Computer Sciences Department
University of Wisconsin -- Madison
Madison, Wisconsin 53706
U. S. A.

Abstract

This paper describes a new scheduling algorithm for a multicomputer connected in a point-to-point fashion. The algorithm is both distributed (every processor runs the same algorithm) and stable (collective load balancing decisions will not cause unnecessary overloading of a processor in the network). We assume that process execution times are not known in advance and that inter-processor transfer times are non-trivial. Load is balanced by migrating suitable jobs after they have been run long enough to obtain an estimate of their required service time. The algorithm is suitable for job scheduling in a distributed time-sharing system implemented on a multicomputer. Performance of the algorithm is investigated through simulation.

1. Introduction

Classical operating systems make job scheduling decisions either by using some a priori estimate of the resources that each job will require (usually supplied by the user) or by collecting statistics during the execution of each job to predict its overall resource requirements. In interactive systems, initial estimates are generally unavailable, so gathered statistics are used instead.

Recently, several researchers have been designing and implementing distributed operating systems for multicomputers^{1,2,3}. (We distinguish a multicomputer, in which processors communicate only by sending messages, from a multiprocessor, in which processors share memory.) These operating systems assign newly arrived jobs to processors based on fairly simple heuristics (such as availability of memory or explicit request of the parent job). In more complex situations, linear programming or network flow algorithms^{4,5,6,7} have been proposed as methods of determining optimal, static assignments of jobs to processors given that the resource needs of all jobs are known

in advance.

Dynamic assignment of jobs to processors has not received as much attention, in spite of the fact that it can increase throughput and decrease response time by taking advantage of loading differences between processors^{8,9}. However, even dynamic job assignment is not a complete solution to the problem of scheduling interactive tasks on a multicomputer. Since properties of a job are not known until its behavior has been sampled by letting it run, a job must be able to start on one machine and then migrate to a more suitable machine as its resource requirements become evident.

Migration is by no means an easy task. Not only must the code and data be moved, but also all logical communications paths must be rerouted. Independent of processor load questions, communication lines may also suffer from overload and may create bottlenecks, and processes do not necessarily function equally well on any machine of the multicomputer. These aspects of migration will be ignored in this paper. We will concentrate on migration purely for the purpose of balancing processor load.

The algorithms that we present are both distributed and stable. A distributed algorithm is important to insure robustness, to prevent information bottlenecks, and because a centralized scheduling algorithm is antithetical to the concept of a distributed computer system. A scheduling algorithm is stable if all job assignment decisions are correct for at least the short term; an algorithm would be unstable if a job continually migrates around the network without accomplishing any useful work. This form of fruitless migration is the multicomputer analog of thrashing on a virtual memory system⁷; we call it processor thrashing.

Processor thrashing can occur on a multicomputer because scheduling decisions made by a processor are based on relatively old data (due to transmission delay

between processors) and are made independently of decisions made by other processors. Thus if processors A and B both observe that processor C is idle, they can offload so much work to processor C that it is now overloaded in relation to processors A and B. Jobs then can be shunted back to processors A and B where the entire cycle can begin again.

Bidding¹⁰ is the best-known example of a distributed scheduling algorithm. However, bidding requires that the communications medium of the multicomputer allow broadcast messages and that each processor maintain a current "price" for resource use. This price is used to answer "requests for quotes" sent by other processors. Adjusting this price in relation to resource use is not straightforward¹¹. Our algorithm requires neither of these restrictions. Migration decisions are based on minimizing the estimated response time for individual jobs; if it is estimated that a job would have a quicker response on a nearby processor (the estimate includes transfer time delay) then it is sent to that processor to be run.

To clarify the scope of our discussion, we state the following working assumptions:

1. All processors have the same speed and are equally capable of running any job. In particular, we ignore the cost of communication with such entities as files, which may differ as jobs move from sites close to the files to sites farther away.
2. The runnable set of processes is scheduled independently on each processor according to a round-robin algorithm.
3. A process may be migrated from any processor to any of its neighbors at any time. The communications cost of this transfer is proportional to the size of the job (the amount of memory it is using) plus communications-network queueing delay.
4. Process migration is never constrained by memory size. We assume either that each processor has ample memory, or that backing storage can be used at each processor to hold those jobs that cannot fit into main memory.
5. The scheduling algorithm that decides when and how to transfer processes requires no overhead. However, messages to achieve cooperation between processors do

entail time overhead.

6. The service time for each process is defined as the amount of time that process spends in the running state from its start until its termination. The service time is not known by the scheduling algorithm, although it may keep historical information that allows it to estimate the job service time distribution.
7. The processors are interconnected by two-way point-to-point communications lines, forming a graph with processors as vertices and communications lines as edges. This graph is not necessarily complete, but it is connected.
8. New jobs can arrive at any machine on the network.

The results reported here are preliminary and based on simulation; we intend to implement the most successful variation on the Arachne operating system^{2,12} to validate these simulations. The algorithm we will discuss can be decomposed into two parts, which we call the load estimation method and the cooperation method. The following sections deal with these methods in detail; we then discuss the combined algorithm and simulation results.

2. Load estimation

In a single processor operating system, the remaining service-time estimate for any job is commonly based on the heuristic that if a job has run a long time, it is likely to run some more, but an actual remaining processing-time estimate is not made. Instead, the job is placed on a queue shared by all jobs of a similar processing-time history. For example, multilevel feedback schedulers place a job on queue k if it has run for k quanta. Jobs on queues with higher k have lower priority. However, in a distributed system, an explicit estimate of the remaining processing time needed by a job is required so that its response time may be estimated under the assumption that it is moved to a different machine in the network.

2.1 Estimating Remaining Processing Time

We consider four algorithms for estimating how much longer a job will continue based only on the processing time it has used to date. The simplest is the memoryless method, which assumes that all jobs have the same expected remaining time, independent of time used so far. Another simple algorithm is pastrepeats; it estimates that remaining time is equal

to time used so far. If we know the distribution of service times, the associated distribution method estimates that the job's remaining processing time is the expected remaining time conditioned by the time already used. For purposes of comparison, the optimal method (which is infeasible in practice) gives the actual remaining service-time requirement.

If we let t be the amount of time used so far by a job, $R(t)$ be the remaining time needed given that t seconds have been used, S represent the service-time random variable (with density $f(s)$, distribution $F(s)$, expected value $E\{S\}$), and $R_E(t)$ the scheduler's estimate of $R(t)$, then these methods estimate $R(t)$ by the following expressions:

memoryless: $R_E(t) = E\{S\}$
(independent of t)

pastrepeats: $R_E(t) = t$

distribution: $R_E(t) = E\{S-t \mid S > t\}$

optimal: $R_E(t) = R(t)$

As a special case, the memoryless method is the distribution method for the exponential distribution. The first step in comparing these methods is the following theorem:

Theorem. If the service distribution is known, then the distribution method has minimal RMS error when averaged over the lifetimes of all jobs.

Proof. Given that $S=s$, the mean square error of an estimation method over the lifetime of a particular job is

$$E\{\text{error} \mid S=s\} = \int_0^s (R_E(t) - (s-t))^2 dt$$

Thus the overall mean square error is

$$E\{\text{error}\} = \int_0^\infty E\{\text{error} \mid S=s\} f(s) ds.$$

Substituting the expression for $E\{\text{error} \mid S=s\}$ into the second equation above, reversing the order of integration, and expanding the squared term yields:

$$\begin{aligned} E\{\text{error}\} &= \int_0^\infty (1-F(t)) [R_E(t)^2 \\ &\quad - 2 R_E(t) E\{S-t \mid S \geq t\} \\ &\quad + E\{(S-t)^2 \mid S \geq t\}] dt \end{aligned}$$

If for every t we select $R_E(t)$ so as to minimize the expression in square brackets,

etc, it is clear that we will minimize the integral. Application of elementary calculus shows that setting

$$R_E(t) = E\{S-t \mid S \geq t\}$$

will minimize the integrand for each t . QED.

Even though this value of $R_E(t)$ minimizes the integral, the size of the expression

$$\int_0^\infty (1-F(t)) E\{(S-t)^2 \mid S \geq t\} dt$$

sets a lower bound for $E\{\text{error}\}$. Since this term is proportional to the variance of S , there may be considerable error in the estimate even in this best case.

Furthermore, the distribution method is difficult to apply in practice. In most situations, the service-time distribution is not known and must itself be estimated. Without an analytic formula for $f(s)$, $R_E(t)$ must be evaluated numerically from the empirical service-time density, at considerable expense in both time and space. We will return to the problem of selecting a good estimation method for remaining service time after discussing how such a method can lead to estimates of response time.

2.2 Estimating Response Time

The primary performance measure for any interactive computer system is response time. It is therefore natural for our scheduling policy to attempt to minimize overall response time. We have assumed that jobs on each processor are scheduled according to a round-robin discipline; for computational simplicity we now assume that the quantum size is small enough that the behavior of the round-robin scheduler can be modeled by processor sharing¹³.

Let $J(P)$ be the set of jobs present at processor P , and let k (not in $J(P)$) be a potential migrant to processor P whose response time we wish to estimate should it move to P . We always know the used processor time t_i for each job i ; this figure could come from updating each job's used processor time at the end of each quantum of service devoted to it. Then $RSP(k, J(P))$, the estimated response time of job k at processor P , can be calculated according to the following algorithm:

```

Algorithm ESTRESPONSE
{ calculate an estimate RSP(k,S) of
the response time for job k if it
were located at a processor with the
set of jobs S }
R := RE(tk);
for all j in S do
begin
  if RE(tj) < RE(tk)
  then R := R + RE(tj)
  else R := R + RE(tk);
end;
RSP(k,S) := R;

```

To understand why ESTRESPONSE provides a good estimate of the job's response time, let us suppose that the estimation method is optimal. Then we claim that RSP(k,J(P)) is the true response time of job k provided that no new jobs arrive at processor P during the execution of job k. All jobs j in J(P) with R_E(t_j) ≥ R_E(t_k) will terminate after job k, so they will be present in the system throughout job k's execution. Therefore if we reduce the remaining execution times of all such jobs to that of job k we will not modify job k's response time in any way. The total length of the modified schedule is the same as the response time of job k. Since the scheduling policy is work-conserving, the length of the schedule is the same under FCFS as it is under PS. The algorithm correctly calculates the length of this schedule.

For the purposes of job migration we will want to extend the algorithm ESTRESPONSE to handle the case where k is in S. The extension skips the iteration of the for loop in the case j=k.

RSP(k,J(P)) estimates the response time of job k at processor P given that job k has arrived at processor P. The total response time of job k at processor P is RSP(k,J(P)) plus the time required to transfer job k to processor P. We define TOTRSP(k,P,Q) as the total response time of job k at processor Q given that job k is presently at processor P. Formally,

$$\text{TOTRSP}(k,P,Q) = \text{TRANSFER}(k,P,Q) + \text{RSP}(k,J(Q))$$

where TRANSFER(k,P,Q) is the time required to transfer job k from P to Q.

We have assumed that migrations are "local" in the sense that a job will only be migrated to a neighboring processor, so P and Q must be immediate neighbors (i. e. there is a direct communications link from P to Q). TRANSFER(k,P,Q) can therefore be calculated from the speed of the communications device and the sum of the sizes of

job k and all of the other migrants in the communications queue from processor P to Q. Since this queue presumably resides in P's main memory, calculation of TRANSFER(k,P,Q) is straightforward.

2.3 A Practical Estimation Method

As mentioned above, while the distribution method of calculating R_E(t) is optimal in a certain sense, it is difficult to implement in the absence of an analytical formula for f(s). We therefore conducted some preliminary simulations to estimate the accuracy of RSP(k,J(P)) when the estimation methods memoryless, distribution, and pastrepeats were used. We used an M/G/1-PS queueing system as the test case. Our accuracy measure was the time-averaged root-mean-square error between RSP(k,J(P)) under the test estimation methods and RSP(k,J(P)) based on the optimal method. Observations of the error were taken for each job in system at each job departure instant. We summarize these statistics in Table I.

Estimation Method	Service Time Distribution			
	uniform low	uniform high	hyper low	hyper high
memoryless	18.4	41.8	33.9	47.1
distribution	6.8	14.4	33.8	55.7
pastrepeats	12.5	23.6	36.7	50.2

Table I
RMS Error in RSP(k,J(P))

The service-time distributions we used were uniform(0.5,6.5) and a two-stage hyperexponential with mean 3.5 and cv² (squared coefficient of variation = VAR[S]/E[S]²) of 3.0. Two values for the arrival rate were used to provide statistics at low and high loadings of the system (utilization = 0.70 and 0.91 respectively).

We see from the table that in comparison to the other methods, the memoryless method performs well for the hyperexponential distribution, but poorly for the uniform distribution. This result can be attributed to the approximate exponentiality of the hyperexponential distribution. Because the memoryless method is dependent on the form of the service-time distribution, it seems unsuitable for our use. The distribution method has the best overall accuracy, but the pastrepeats method is not much worse. Because of the ease of implementation of pastrepeats com-

pared to the distribution method, we use pastrepeats in our distributed scheduling policy.

2.4 A Total Load Estimate

For the purposes of the cooperation policy as described below, it will be convenient to have a single number that represents the total load at a particular processor. The total number of jobs present at a processor is unsuitable for such an estimate since the true load could vary widely depending on the remaining service times for those jobs. On the other hand, the sum of all remaining service times does not provide an accurate picture without including the number of jobs.

To illustrate these problems, suppose processor A has one job with a true remaining service time of 200 seconds, while processor B has 100 jobs each with a true remaining service time of 1 second. Then the response time of a new one-second job sent to processor A would be two seconds, while if the job were sent to processor B its response time would be 101 seconds (assuming no other new arrivals). On the other hand, the response time of a new 200-second job at processor A would be 400 seconds, while at processor B it would be 300 seconds.

As a composite measure of these effects we define the load at processor P as $RSP(k, J(P))$ where k is a job whose remaining service time is equal to the average overall service time. All processors in the network use the same value for this average. For example, if the average service-time requirement is 20 seconds, then the load on the one-job processor mentioned above is 40 seconds; the load on the 100-job processor is 120 seconds. Since the expected response time is always equal to or greater than the average service time, we subtract the average service time to yield the expected delay measure.

The expected delay may be used to decide if any job at all should be migrated between two adjacent processors. If they have very different expected delays, some job on the more heavily loaded processor should be chosen for migration.

3. Cooperation Policy

Processors must communicate in order to reach decisions on migration. This communication may be structured in various ways. At one extreme, each machine can periodically broadcast its load estimate. This method leads to very large overhead, since each machine periodically hears messages from every other machine. It is also difficult to match processors together, but if processors are not matched,

then one very underloaded machine can suddenly receive migrant jobs from many sources, turning it into a very overloaded machine. Soon this processor will try to send some of those jobs elsewhere, leading to processor thrashing.

Since we restrict migration to follow the point-to-point communications lines of the computer network, unlimited broadcast may be restricted to direct-neighbor broadcast. The problem of processor thrashing is still present, as is the large number of messages needed.

An elegant means of structuring communication is to build temporary pairings between processors. One algorithm has been given by Finkel¹⁴ for constructing permanent pairs. A modified version of the algorithm has this outline: Each processor asks some randomly chosen direct neighbor if it will pair. While awaiting an answer, the querier rejects any queries from other neighbors. If it receives a rejection, it again picks a random neighbor and tries again. If it receives a query from its own intended mate, then a pair is formed. The pair is broken at the mutual agreement of the two mates, for example, after a job has been migrated. It is not necessary that both break from the pair simultaneously. During the time that a pair is in force, both mates reject other queries.

Bernstein has proposed a related algorithm for pairing¹⁵. He assigns identification numbers to each processor and allows processors to remember who has queried them. In one simple variant, a processor that would otherwise reject a query will instead postpone a decision if the querier has a larger identification number than his own. In addition, the identification number of the querier must be higher than the identification number of any other querier that has been postponed; this rule prevents cycles of postponement. If there is a current postponed query and a new query is postponed instead, the original querier gets a rejection. At the time the postponder either gets a rejection itself from its intended mate or breaks from a pair, it may immediately pair with the postponed querier (by sending it a query).

Simulations were run to compare Bernstein's identification number method with the other method, which we will call the simple method. All tests are performed on a square array (5x5 processors) connected in a mesh (each internal processor has four direct neighbors). The processors were given identification numbers in row-major order. The results may be summarized as follows:

1. Both methods are fairly insensitive to the message delay distribution, be it exponential, Erlangian, or uniform (with a small window) about the mean. We therefore chose a uniform message delay distribution for computational simplicity.
2. If pairs last time p , average message passing time is m , and average waiting time for achieving pair is w , then when $p \gg m$, the identification-number method yields $w = p/2$. The simple method takes up to 80% longer (and gets up to 80% fewer pairs per second), with the difference greatest when p is small with respect to m .

The purpose of migration is to disperse load evenly across the network. We tested the ability of the simple and identification number methods to disperse load by placing 100 tokens on some processor. When processors pair, tokens are moved between them to equalize the count on the two processors. Such a pairing lasts a time equal to the number of tokens transferred. The tokens therefore model migrating jobs. A measure of the current dispersion is the variance of the number of tokens across all processors; this figure reaches 0 after enough time has elapsed.

Depending on which processor is given the 100 tokens, the identification number method either performed significantly better or significantly worse than the simple method. The best performance was achieved when processor 1 gets the initial tokens; worst performance was elicited when processor 25 got them. This asymmetry is due to the fact that a low-numbered processor usually has a postponed querier at the time it breaks a pair or gets a rejection; a high-numbered processor usually does not have such a postponement. Therefore, lower-numbered processors form pairs more quickly and are able to disperse their tokens more efficiently.

In order to remove the asymmetry, postponements were decided on the basis not of identification number, but rather preferring to postpone the querier whose number of tokens is most different from the postponder's. This method, which we call the absolute difference method, yielded superior dispersion. However, it is possible for this method to enter a deadlock, although this situation was never observed. A deadlock may be formed of a cycle of four processors, two of which have 0 tokens, and the other two have one token. If each has queried its clockwise neighbor, every query will be postponed.

To insure absence of deadlock, it is sufficient to include a timeout on postponement, so that any processor that is postponing a query for more than a given amount of time sends a rejection instead. An alternate method is to use a signed difference method, preferring to postpone the querier whose number of tokens is least (and less than the postponder's token count). This method yields very good dispersion if the number of tokens placed on the first processor is positive, and adequate (but inferior) dispersion if the initial number is negative. Intuitively, this method will quickly remove jobs from overloaded processors, but is not so quick to add them to underloaded ones.

Figure 1 (placed at the end of the paper) presents a graph of dispersion verses simulated time for the simple, identification number, absolute, and signed methods. On the basis of these data we chose the signed method for our distributed load-balancing algorithm.

4. The load-balancing algorithm

The cooperation policy may be combined with the load estimation policy to create an overall load-balancing algorithm. Current load estimates are used in the signed pairing algorithm to form pairs that differ greatly in load. Once a pair is formed, the more heavily loaded processor decides which jobs, if any, to send to the other processor, based on greatest expected improvement in the response ratio of the migrant jobs. A processor only tries to find a mate if it has at least two jobs; otherwise, migration from this processor is never reasonable. However, every processor is willing to respond favorably to a query.

Several variants of this algorithm have been tested by simulation. One result we quickly found was that fruitless pairings and attempts were very frequent. A pairing is fruitless if no job can be migrated. An attempt is fruitless if a query is rejected. These fruitless actions would also cause overhead in an actual implementation. To reduce this overhead, we introduced a relaxation period after a processor has queried all of its neighbors. During this period, no new attempts at pairing are made. Arrival of a new job or a query terminates relaxation early. We also changed the pairing algorithm so that only the first neighbor queried is chosen at random; when a reply is received, the next neighbor in turn is asked until all neighbors have been queried. This variant of the pairing algorithm is a distributed analog of polling in a centralized system.

We also found that in order to produce a reasonable number of migrations, the time a processor remains paired must be as short as possible and multiple migrants per pairing must be allowed. Processors cannot remain paired during the entire period they are exchanging jobs, and the actual migration of jobs must be as rapid as possible. In order to accomplish these goals, we assume that each processor maintains two job lists. One list contains the jobs presently residing at the processor; the other contains jobs being sent to the processor that have not yet been received. As jobs are received, they are removed from the second list and placed on the first.

Every query sent from the processor includes both job lists; the time consumed by each job is also included. For the purpose of calculating the load or response time at a processor, the set $J(P)$ is taken as the union of the two lists.

The list of migrating jobs is updated as follows: The sending processor always decides which jobs to send based on its current load and its mate's load (as of the last query from its mate). The sender sends a list of migrant jobs before the first migrant itself is sent. The receiving processor remains paired just long enough to receive this list. A processor can therefore never become overloaded by receiving migrants from more than one source, nor can a group of jobs in transit cause a particular processor to become overloaded; the destination processor will always know about them before they are sent. In this way the pairing algorithm allows stable distributed scheduling decisions to be made.

To allow more than one migration per pairing is straightforward. Let us suppose that processors A and B are paired, and let us assume that A has a larger load. Then A has two sets of jobs to consider: $J(A)$ and C , where C is the copy of $J(B)$ that processor B sent to A. Now for each job j in $J(A)$, processor A calculates the ratio

$$I_S(j) = \frac{R_E(j, J(A))}{R_E(j, C) + \text{TRANSFER}(j, A, B)}$$

This ratio represents the improvement in service that job j can receive at processor B. If $I_S(j) < 1$ for all j in $J(P)$ then no migrant is sent. A sends B a message indicating this fact and the pair is broken. On the other hand, if $I_S(j) > 1$ for one or more jobs, the job with the largest service improvement is selected as the first migrant. To allow

for the eventual presence of this job at B, j is removed from $J(A)$ and a copy is inserted into C . The actual job j is removed from the ready set and placed in the communications queue for processor B. The entire procedure is repeated to determine if there are other jobs to migrate. When this algorithm terminates with $I_S(j) < 1$ for all j , processor A formulates a message listing the migrants to be sent to B, puts the message at the head of the communications queue and starts the transfer. Processor A immediately breaks its pairing with processor B, and if it has not yet polled all of its neighbors, begins sending a query to its next neighbor in turn. Otherwise processor A relaxes.

5. The Distributed Scheduling Algorithm

Before describing the results of our simulations, we present a concise description of the algorithm. Each processor can be in one of three states: idle, pairing and migration. A processor is idle if it has less than two jobs or is relaxing. Processors cooperate by sending one of three types of messages: queries, rejections, and migrations.

A processor enters the pairing state when a second job arrives, when it receives a query from a neighbor, when its relaxation period expires, or when it leaves the migration state and has not yet queried all of its neighbors. In the pairing state, processors try to form pairs by cyclically querying each neighbor; at the end of each cycle, the processor relaxes. The first neighbor in each cycle is chosen at random. Each query message includes a list of local jobs, both those present and those in migration. This information is used to compute the querier's load for purposes of pairing.

A processor in the idle state responds to a query with a query, forming a pair. Processors in the pairing state respond with a query only if queried by their intended mate; otherwise, they respond with a rejection message. Processors in the migration state respond with rejection. Rejections may be postponed if the querier has a lower load than the rejector and that load is lower than the load on a postponed querier. Only one querier may be postponed at a time; any other querier gets an immediate rejection.

Once a pair is formed, the mates enter the migration state. The member with the larger load selects jobs to migrate to the other and sends a list of those jobs, followed by the jobs themselves, if any. As soon as this decision has been made, the sender breaks the pair and returns to the pairing state (or the

idle state, if it is time to relax). When the recipient receives the list, it also breaks the pair and returns to the pairing state (or the idle state).

The jobs to be migrated are selected by comparing their expected time to complete on their current host with the expected time to complete on its mate. Migration delay is included in this estimate. The job with the best ratio of service time on the mate to service time on the current host is selected to be sent first. Decisions for other jobs are based on the assumption that the first job has been received by the mate.

6. Simulation Results

These experiments were also performed on a square mesh of 25 processors. The system arrival process was assumed to be Poisson with intensity equal to 80% of 25 times the per-processor service rate. The total system utilization was therefore 0.8. Various arrival patterns were tested to see how well the migration algorithm adapted to local imbalance. For example, in one test case all jobs were initially assigned to an arbitrary processor chosen at random, while in another test case all arriving jobs were assigned to odd-numbered processors.

The message delay time was set to 0.10 seconds (a reasonable figure on Arachne), and the service time distribution was a two-stage hyperexponential distribution with $CV^2=3.0$ and a mean of 1.0. (It is well known that CPU service time distributions are more skewed than exponential; a hyperexponential distribution is a convenient representation of this fact¹⁶). Rather than simulate a round-robin schedule, the simulation actually implements a processor-sharing scheduler. The simulations were written in SIM-PAS^{17,18}, a simulation extension of the language PASCAL.

6.1 A Relaxation Time Experiment

In our first experiment, we tested various settings of the relaxation time to see how it affected the quality of the load balancing. We measure this quality by the average response ratio perceived by all jobs that terminate. In this test, all new jobs were assigned to odd-numbered processors chosen at random according to a uniform distribution. Table II shows the statistics for the first 100 seconds of

simulated time.

relax time	pair- ings	termi- nations	migra- tions	resp. ratio	msgs sent
0.01	4961	1960	819	4.09	16760
0.10	4760	1953	836	4.14	15317
0.25	4507	1949	797	4.09	13972
0.50	4138	1948	820	4.33	12928
0.75	3905	1941	828	4.46	12115
1.00	3696	1949	797	4.40	11595
2.00	3307	1928	780	4.70	10889

Table II
Scheduling Algorithm Performance
For Different Relaxation Intervals
Odd Numbered Processors Loaded

These results show that the algorithm is relatively insensitive to the relaxation interval until the relaxation interval becomes large in relation to the average job inter-arrival time. Since changing the relaxation interval from 0.01 to 0.5 reduces the total number of messages sent by more than 20%, we have elected to use a relaxation interval of 0.5 seconds. Although the number of messages given in the table may seem excessive, the number given represents only about 5 messages per processor per second.

6.2 Effectiveness of Load Balancing

To demonstrate the effectiveness of load balancing in our algorithm, we discuss two additional simulation runs.

In the first run each arriving job is assigned at random according to a uniform distribution to one of the 25 processors. Since the simulation implements a processor-sharing scheduler and the arrival process is Poisson, it follows that without migration the mean number of jobs at each processor would be the same as for an M/M/1 queueing system¹³ with utilization=0.8. Thus

$$\begin{aligned}\text{mean number of jobs per processor} &= \rho / (1-\rho) \\ \text{mean time in system} &= 1 / (\lambda * (1-\rho))\end{aligned}$$

If there is no migration, the mean number of jobs per processor should be 4; the mean time in system should be 6.25 seconds. The response ratio should therefore be about 6.25 (since the average service time is 1 second). Table III gives simulation results for our algorithm during each 25 seconds of a 100 second run. (The statistics are reset at the end of each 25 second interval.)

time (seconds)	response ratio	average queue size
25	2.29	1.33
50	3.29	2.23
75	3.33	2.50
100	3.29	2.47

Table III
Migration Results
All Processors Loaded

These data show the clear superiority of migration even when the load is very uniform.

For our final example we consider a case in which all arrivals to the system are initially assigned to one of four processors. Figure 2 shows the location of the loaded processors in the network, with the id numbers for the loaded processors enclosed in square brackets.

1	6	11	16	21
2	7	[12]	17	22
3	[8]	13	[18]	23
4	9	[14]	19	24
5	10	15	20	25

Figure 2
Loaded Processors for 4-Processor Example

Since the processor interconnections are along horizontal and vertical grid lines only, processor 13 is surrounded by heavily loaded processors. We conjecture that a less sophisticated scheduling algorithm would overload processor 13. In fact, our simulations show that while the system as a whole is overloaded (the loaded processors simply cannot offload jobs fast enough), 168 jobs are sent to processor 13 during 100 seconds while only 8 jobs are sent back from processor 13 to one of the loaded processors.

7. Concluding Remarks

We have described a new distributed load balancing algorithm in which processors cooperate in making load balancing decisions but do not depend on a centralized controller. Although the algorithm is complex, we believe that the stability of this algorithm is worth the complexity. The experience we have gained in studying this algorithm through simulation will help us when we install it in the Arachne multicomputer system.

We are currently investigating simpler load migration schemes in order to evaluate the importance of stability in load balancing algorithms of this kind.

8. Acknowledgements

Computer time for the simulations described in this paper was provided in part by the Madison Academic Computing Center. R. M. Bryant was supported in part by the National Science Foundation through grant MCS-800-3341.

9. References

- [1] D. Mills, "An Overview of the Distributed Computer Network," Proceedings of the National Computer Conference 45, AFIPS Press, pp. 523-531 (1976).
- [2] M. Solomon and R. Finkel, "The Roscoe Distributed Operating System," Proceedings of the Seventh Symposium on Operating Systems Principles, pp. 108-114 (December 1979).
- [3] L. D. Wittie, "Micronet: A reconfigurable microcomputer network for distributed systems research," Technical Report 143, Department of Computer Science, State University of New York at Buffalo (April 1978).
- [4] V. Balachandran, "An Integer Generalized Transportation Model for Optimal Job Assignment in Computer Networks," Operations Research 24, 4, pp. 742-759 (July-August 1976).
- [5] H. L. Morgan and K. D. Levin, "Optimal Program and Data Locations in Computer Networks," Communications of the ACM 20, 5, pp. 315-321 (May 1977).
- [6] H. S. Stone, "Multiprocessor Scheduling with the aid of Network Flow Analysis," IEEE Transactions on Software Engineering SE-3, 5, pp. 315-321 (May 1977).
- [7] H. S. Stone, "Control of Distributed Processes," IEEE Computer, pp. 97-106 (July 1978).
- [8] A. K. Agrawala, S. Tripathi, and G. Ricart, "Adaptive Routing Using a Virtual Waiting Time Technique," Technical report TR-817, University of Maryland Computer Science Department (November 1979).
- [9] Yuon-Chieh Chow and K. H. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," IEEE Transactions on Computers C-28, 5, pp. 354-361 (May 1979).

- [10] D. J. Farber, J. Feldman, F. R. Heinrich, M. D. Hopwood, K. C. Larson, D. C. Loomis, and L. A. Rowee, "The Distributed Computing System," Proceedings of the Seventh Annual IEEE Computer Society International Conference, pp. 31-34 (February 1973).
- [11] Carl M. Ellison, "The Utah TENEX Scheduler," Proceedings of the IEEE 23, 6, pp. 940 - 945 (June 1975).
- [12] R. A. Finkel, M. H. Solomon, and R. Tischler, "Arachne User Guide, Version 1.2," Technical Summary Report #2066, University of Wisconsin Mathematics Research Center (April 1980).
- [13] E. G. Coffman and P. J. Denning, Operating Systems Theory, Prentice-Hall (1973).
- [14] R. A. Finkel, M. H. Solomon, and M. L. Horowitz, "Distributed algorithms for global structuring," Proceedings of the National Computer Conference 48, AFIPS Press, pp. 455-460 (June 1979).
- [15] A. J. Bernstein, "Output guards and nondeterminism in 'Communicating sequential processes'," Transactions on Programming Languages and Systems 2, 2, pp. 234-238 (April 1980).
- [16] P. Brinch Hansen, Operating Systems Principles, Prentice-Hall, Inc. (1973).
- [17] R. M. Bryant, "SIMPAS -- A Simulation Language Based on PASCAL," Technical Report #390, University of Wisconsin-Madison Computer Sciences Department (June 1980) Presented at the 1980 Winter Simulation Conference, December 3-5, 1980, Orlando, Florida.
- [18] R. M. Bryant, "SIMPAS User Manual," Technical Report #391, University of Wisconsin-Madison Computer Sciences Department (June 1980).

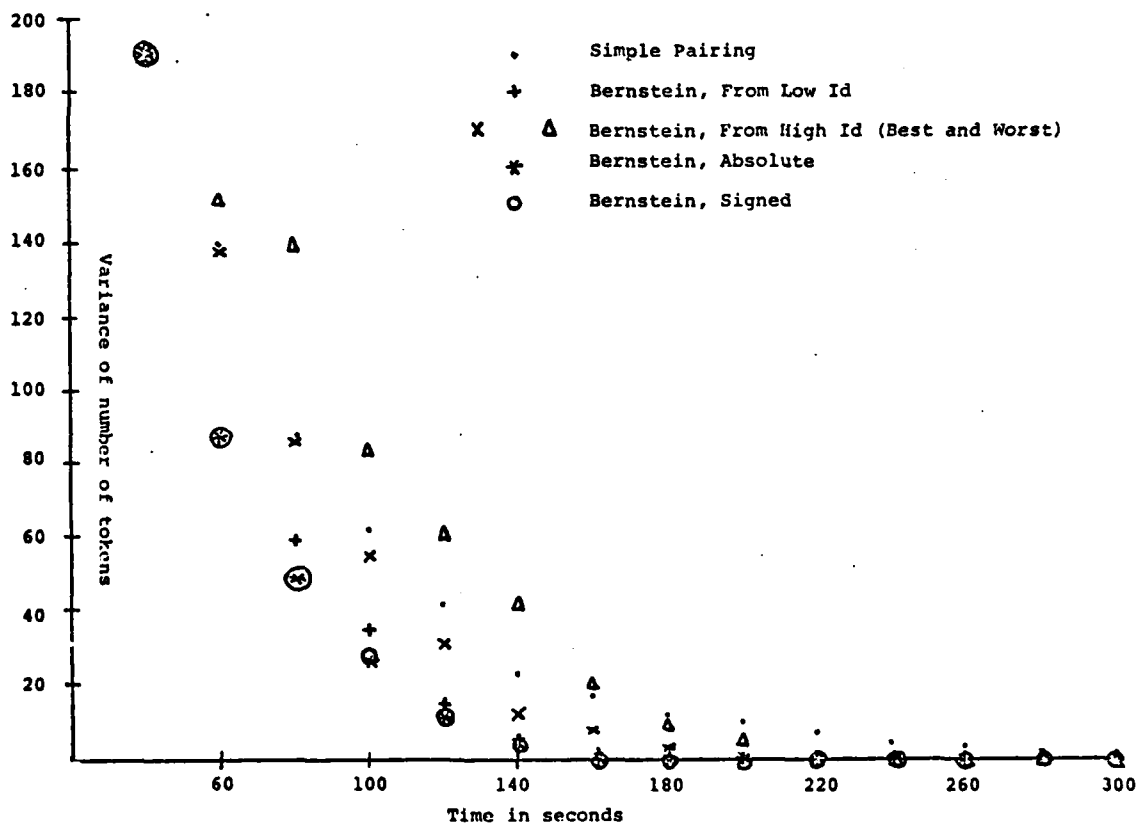


Figure 1
Performance of Pairing Methods

APPENDIX E

Parallelism in Alpha-Beta Search

Raphael A. Finkel
John P. Fishburn

Journal of Artificial Intelligence
Vol. 19, No. 1 (Sept. 1982)

"This article originally appeared in the journal Artificial Intelligence, Number 19 (1982) pp. 89-106, published by the North-Holland Publishing Company, P.O. Box 1991, 1000 BZ AMSTERDAM, The Netherlands".

Parallelism in Alpha-Beta Search

Raphael A. Finkel

Computer Sciences Department
University of Wisconsin-Madison

John P. Fishburn

Bell Laboratories
Murray Hill, New Jersey

ABSTRACT

We present a distributed algorithm for implementing α - β search on a tree of processors. Each processor is an independent computer with its own memory and is connected by communication lines to each of its nearest neighbors. Measurements of the algorithm's performance on the Arachne distributed operating system are presented. A theoretical model is developed that predicts at least order of $k^{1/2}$ speedup with k processors.

Parallelism in Alpha-Beta Search

Raphael A. Finkel

Computer Sciences Department
University of Wisconsin-Madison

John P. Fishburn

Bell Laboratories
Murray Hill, New Jersey

1. INTRODUCTION

The α - β search algorithm is central to most programs that play games like chess. It is now well-known¹ that an important component of the playing skill of such programs is the speed at which the search is conducted. For a given amount of computing time, a faster search allows the program to "see" farther into the future. In this paper we present and analyze a parallel adaptation of the α - β algorithm. This adaptation, which we call the *tree-splitting algorithm*, speeds up the search of a large tree of potential continuations by dynamically assigning subtree searches for parallel execution.

In Section 2, we review the α - β algorithm. Section 3 discusses parallel implementations of the α - β algorithm suggested by other workers. Section 4 formally describes the tree-splitting algorithm. Section 5 discusses some possible optimizations and variations of the algorithm. Section 6 presents two sets of performance measurements for this algorithm. One set was taken through simulation, the other on a network of microprocessors. Section 7 develops a theoretical model that predicts speedup for an arbitrary number of processors, and Section 8 compares these predictions with the measurements of Section 6.

2. THE ALPHA-BETA ALGORITHM

We assume that the reader is familiar with the negamax and alphabeta algorithms as described by Knuth and Moore:²

```
function negamax(p:position):integer;
var   m: integer;
      i,d : 1..MAXCHILD;
      succ : array[1..MAXCHILD] of position;
begin
  determine the successor positions succ[1],...,succ[d];
  if d = 0 then { terminal position }
    return(staticvalue(p));
  { find maximum of child values }
  m := -  $\infty$ ;
  for i := 1 to d do
    m := max(m, - negamax(succ[i]));
  return(m);
end;
```



```

function alphabeta(p : position;  $\alpha, \beta$  : integer) : integer;
var   i, d : 1..MAXCHILD;
      succ : array[1..MAXCHILD] of position;
begin
  determine the successor positions succ[1], ..., succ[d];
  if d = 0 then { terminal position }
    return(staticvalue(p));
  for i := 1 to d do
    begin
       $\alpha := \max(\alpha, -\text{alphabeta}(\text{succ}[i], -\beta, -\alpha));$ 
      if  $\alpha \geq \beta$  then return( $\alpha$ ) { cutoff }
    end;
  return( $\alpha$ )
end;

```

The function alphabeta obeys the following important property: For a given position p , and for values of α and β such that $\alpha < \beta$,

if $\text{negamax}(p) \leq \alpha$, then $\text{alphabeta}(p, \alpha, \beta) \leq \alpha$

if $\text{negamax}(p) \geq \beta$, then $\text{alphabeta}(p, \alpha, \beta) \geq \beta$

if $\alpha < \text{negamax}(p) < \beta$, then $\text{alphabeta}(p, \alpha, \beta) = \text{negamax}(p)$.

The first and second cases above are called *failing low* and *failing high* respectively. In the third case, *success*, alphabeta accurately reports the negamax value of the tree. Success is assured if $\alpha = -\infty$ and $\beta = \infty$. The pair (α, β) is called the *window* for the search.

The alpha-beta algorithm is strongly serial: It uses information from one part of the lookahead tree to avoid work in another part. If the lookahead tree is decomposed into several pieces and these pieces searched simultaneously, work that the serial algorithm avoids may be performed. Nevertheless, we will see that such a decomposition can achieve speedup.

Since we will later be referring to a tree of processors, we reserve the following notation for nodes of lookahead trees: A node is often called a *position*. A node's child is its *successor*, and its parent is its *predecessor*. If each interior node has n successors, we say that the tree has *degree* n . The *level* of a node or subtree is its distance from the root.

We define the *speedup* of a parallel algorithm over a serial one to be the time required by the serial algorithm divided by the time for the parallel algorithm. We will restrict our attention to parallel computers built as a tree of serial computers. A node in this tree is a *processor*. A processor's parent is its *master*, and its child is its *slave*. If each interior processor has n slaves, we say that the tree has *fanout* n .

3. RELATED WORK

3.1. Parallel-Aspiration Search

In order to introduce parallelism, Baudet³ rejects decomposition of the lookahead tree in favor of a *parallel aspiration search*, in which all slave processors search the entire lookahead tree, but with different initial α - β windows. These windows are disjoint, and in the simplest variant their union covers the range from $-\infty$ to $+\infty$. Since each window is considerably smaller than $(-\infty, +\infty)$, each processor can conduct its search more quickly. When the processor whose window contains the true negamax value of the tree finishes, it reports this value, and move selection is complete. Baudet analyzes several variants of this algorithm under the assumption of randomly distributed terminal values, and concludes that the obtainable speedup is limited by a constant independent of the number of processors available. This maximum is

established to be approximately 5 or 6. Surprisingly, for k equal to 2 or 3, Baudet's method yields more than k -fold speedup with k processors. Baudet infers that the serial α - β search algorithm is not optimal, and estimates that a 15 to 25 percent speedup may be gained by starting the search with a narrow window.

Since a narrow window does not speed up a successful search when moves are ordered best-first, Baudet's method yields no speedup under best-first move ordering.

3.2. Mandatory-Work-First Search

Akl, Barnard, and Doran⁴ report simulation measurements of a parallel tree-decomposing alpha-beta algorithm. This algorithm distinguishes between those parts of a subtree that must be searched and those parts whose need to be searched is contingent upon search results in other parts of the tree. By searching mandatory nodes first, their algorithm attempts to achieve as many of the cutoffs seen in the serial case as possible.

We have analyzed a parallel algorithm based on these ideas, and will report on it elsewhere.

4. THE TREE-SPLITTING ALGORITHM

We now describe a parallel algorithm for implementing α - β search on a tree of processors. The root processor evaluates the root position. Each interior processor evaluates its assigned position by generating the successors and queuing them for parallel assignment to its slave processors. Thus a processor at level N in the processor tree always evaluates positions at level N in the lookahead tree. As an interior processor receives responses from its slaves, it narrows its window and tells working slaves about the improved window. When all successors have been evaluated (or a cutoff has occurred), the interior processor is able to compute the value of its position. Each leaf processor evaluates its assigned position with the serial α - β algorithm. When a processor finishes, it reports the value computed to its master.

4.1. The Leaf Algorithm

The leaf algorithm runs at leaf nodes of the processor tree. We will describe its interactions with its master by means of remote procedure calls.⁵ The algorithm can also be expressed in a message-passing or shared-memory form. The master calls the function `leaf $\alpha\beta$` (line 19) remotely. A master can interrupt a search in progress to tell its slave of a newly-narrowed window by invoking the asynchronous "update" procedure in the slave (line 3). The variables α and β (line 1) are global arrays, not formal parameters, in order to facilitate updating their values in each recursive call of `alphabeta` when the new window arrives.

Here is the leaf algorithm:

```

1  $\alpha, \beta$  : array[1..MAXDEPTH] of integer;

3 asynchronous procedure update(new $\alpha$ , new $\beta$  : integer);
4   { update is called asynchronously by my master
5   to inform me of the new window (new $\alpha$ , new $\beta$ ) }
6 var tmp : integer;
7   k : 1..MAXDEPTH;
8 begin
9   for k := 1 to MAXDEPTH do
10    begin { update  $\alpha, \beta$  arrays }
11       $\alpha[k] := \max(\alpha[k], \text{new}\alpha)$ ;
12       $\beta[k] := \min(\beta[k], \text{new}\beta)$ ;
13      tmp := new $\alpha$ ;
14      new $\alpha := -\text{new}\beta$ ;
15      new $\beta := -\text{tmp}$ ;
16    end
17 end;

19 function leaf $\alpha\beta$ (p : position;  $\alpha, \beta$  : integer) : integer;
20 begin
21    $\alpha[1] := \alpha$ ;
22    $\beta[1] := \beta$ ;
23   return(alphabeta(p, 1));
24 end;

26 function alphabeta(p:position; depth:integer): integer;
27 var succ:array[1..MAXCHILD] of position; {successors}
28   succno : 1..MAXCHILD; { which successor }
29   succlim : 1..MAXCHILD; { how many successors }
30 begin
31   determine the successors succ[1], ..., succ[succlim];
32   if succlim = 0 then return(staticvalue(p));
33   for succno := 1 to succlim do
34     begin { evaluate each successor }
35        $\alpha[\text{depth}+1] := -\beta[\text{depth}]$ ;
36        $\beta[\text{depth}+1] := -\alpha[\text{depth}]$ ;
37        $\alpha[\text{depth}] := \max(\alpha[\text{depth}], -\text{alphabeta}(\text{succ}[\text{succno}], \text{depth}+1))$ ;
38       if  $\alpha[\text{depth}] \geq \beta[\text{depth}]$  then return( $\alpha[\text{depth}]$ ); { cutoff occurs }
39     end { for succno }
40   return( $\alpha[\text{depth}]$ );
41 end; { function alphabeta }

```

4.2. The Interior Algorithm

The interior algorithm interior $\alpha\beta$ runs on interior nodes of the processor tree. When interior $\alpha\beta$ is activated, it generates all successors of the position to be evaluated (line 25). Each of its slaves is requested to evaluate one of these positions; the remaining positions are queued for later service. This queue is implemented by the parallel for-loop (lines 30 to 42). A separate process is created (line 30) for each successor, and each process attempts to gain exclusive control of a slave processor (line 32). When successors outnumber slaves, some processes must remain blocked within "idle_slave" until slaves can be allocated to them.

Interior $\alpha\beta$ may take various actions when a slave returns. First, if the returned value

causes the current α value to increase, then interior $\alpha\beta$ sends $-\alpha$ as an updated β value to all active slaves (line 39). Second, if α has been increased so that it becomes greater than or equal to β , then an α - β cutoff occurs. The nonpositive-width window is sent to all active slaves, quickly terminating them (line 39). Meanwhile, interior $\alpha\beta$ empties its queue of waiting successor positions. (In the algorithm shown below, this effect is achieved by the test on line 33.) Third, if the queue of unevaluated successor positions is non-empty, the reporting slave is assigned the next position from the queue.

If interior $\alpha\beta$ is interrupted by an update call from its master, it relays this new window to its slaves (lines 3 to 14).

When all successors have been evaluated, interior $\alpha\beta$ returns the final value to its master (line 43). In a game situation, the algorithm at the root node might serve as the user interface, and would remember which move has the maximum value.

Here is the interior algorithm:

```

1 var gl $\alpha$ , gl $\beta$  : integer; { global  $\alpha, \beta$  }
2   q : integer; { depth of processor tree }
3 asynchronous procedure update(new $\alpha$ , new $\beta$  : integer);
4   { update is called asynchronously by my master
5     to inform me of the new window (new $\alpha$ , new $\beta$ ) }
6 begin
7   atomically do
8     begin
9       gl $\alpha$  := max(gl $\alpha$ , new $\alpha$ );
10      gl $\beta$  := min(gl $\beta$ , new $\beta$ );
11    end; { atomically do }
12  for all slaveid do
13    slaveid.update(-gl $\beta$ , -gl $\alpha$ );
14 end; { update }

16 function interior $\alpha\beta$ (p: position ;  $\alpha, \beta$ : integer) : integer;
17 var succ: array[1..MAXCHILD] of position; { successors }
18   succno : 1..MAXCHILD; { which successor }
19   succlim : 1..MAXCHILD; { how many successors }
20   tmp : array[1..MAXCHILD] of integer;
21   function g : integer;
22 begin
23   gl $\alpha$  :=  $\alpha$ ;
24   gl $\beta$  :=  $\beta$ ;
25   determine the successors succ[1], ..., succ[succlim];
26   if succlim = 0 then return(staticvalue(p));
27   if depth(succ[1]) < q then
28     g := interior $\alpha\beta$ ;
29   else g := leaf $\alpha\beta$ ;
30   parfor succno := 1 to succlim do
31     begin
32       slaveid := idle_slave();
33       if gl $\alpha$  < gl $\beta$  then
34         begin
35           tmp[succno] := -slaveid.g(succ[succno], -gl $\beta$ , -gl $\alpha$ );
36           if tmp[succno] > gl $\alpha$  then
37             begin
38               atomically do gl $\alpha$  := max(tmp[succno], gl $\alpha$ );
39               for all slaveid do slaveid.update(-gl $\beta$ , -gl $\alpha$ );
40             end; { if tmp[succno] > gl $\alpha$  }
41           end; { if gl $\alpha$  < gl $\beta$  }
42         end; { parfor succno }
43       return(gl $\alpha$ );
44 end; { interior }

```

5. OPTIMIZATIONS

Since the tree-splitting algorithm can be optimized in several ways, it should be considered the simplest variant of a family of tree-decomposing algorithms for α - β search. As a first optimization, since most of a master's time is spent waiting for messages, that time could be spent profitably doing subtree searches. However, only the deepest masters could hope to compete with their slaves in conducting searches. All other masters are by themselves slower than their slaves because their slaves have slaves below them to help. However, more than

half of all masters control leaf processors, and greater speedup should be achieved by running a leaf algorithm along with these masters on the same processors. We might expect an additional 1.5-fold speedup from this technique.

A second optimization groups several higher-level masters onto a single processor. For example, the 3 highest processors in a binary processor tree could be replaced by 3 processes running on a single processor.

Finally, the root processor may send a special α - β window to the slave working on the last unevaluated successor. This window is $(-\alpha-1, -\alpha)$ instead of the usual $(-\beta, -\alpha)$. If that successor is not the best, then the slave's search will fail high as usual, but the minimal window speeds its search. If that successor is best, then the smaller window causes the search to fail low, again terminating faster. In either case, the root processor determines which successor is the best move, even though its value may not be calculated. By speeding the search of the last successor, the idle time of the other slaves is reduced. (This narrow window given to the root's last subtree search can also be used in serial α - β search.)

We can generalize this technique in the following way, called *alpha raising*: Suppose that each successor of the root is being evaluated by a different slave, and that *slave*₁'s current α value, α_1 , is lower than any other, and that *slave*₂ has the second lowest α value, say α_2 . Update α_1 to α_2-1 , speeding up *slave*₁. If this update causes *slave*₁'s otherwise successful search to fail low, then the reported value is still lower than all others, and that move is still discovered to be best.

6. MEASUREMENTS OF THE ALGORITHM

Two sets of measurements were taken. The first set was taken on a network of LSI-11 microcomputers running under the Arachne† operating system.⁶ The second set was taken by simulation.

The game of checkers was used to generate lookahead trees. Static evaluation was based on the difference in a combination of material, central board position for kings and advancement for men. Moves were ordered best-first according to their static values. General α -raising was not employed, except for the special case for the last successor. Ten board positions were chosen for use in these experiments. These positions actually arose during a human-machine game; they span the entire game. All lookahead trees from these positions were expanded to a depth of 8.

6.1. Measurements on Arachne

A single LSI-11 machine searches lookahead trees at a rate of about 100 positions per second. Inter-machine messages can be sent at a rate of about 70 per second. Only 5 processors were available in Arachne at the time of these experiments, so it was not possible to use Arachne to test processor trees of height greater than one. Each of the ten board positions was evaluated with the serial algorithm, and with processor trees of height one and fanout two and three. For each processor tree of height q , fanout f , and $k = f^q$ leaf processors, Table 1 gives the minimum, average, maximum, and standard deviation of the speedups in evaluating the ten lookahead trees.

† We have changed the name of the Roscoe distributed operating system to Arachne, since Roscoe is a registered trademark of Applied Data Research, Incorporated.

q	f	k	min	avg	max	std
1	2	2	1.37	1.81	2.36	0.31
1	3	3	1.37	2.34	3.15	0.56

Table 1. Speedup on Arachne.

Surprisingly, more than k-fold speedup was occasionally achieved with k slaves: Three out of the ten positions were sped up by more than 2 with 2 slaves, and two of those three were sped up by more than 3 with 3 slaves. In each such case, subtree evaluations finished in a different order than they were assigned. While one large subtree was being evaluated by one slave, another smaller subtree was assigned and finished. The large subtree's evaluation then received a call on "update" that sped it up or even terminated it.

6.2. Simulation Measurements

Binary and ternary processor trees of depth one, two, and three were simulated on the UNIX[†] operating system. Processors were simulated with processes, communications lines with pipes. Within the simulation, the time for evaluation of one terminal position was set at ten units of time. The time for remotely calling a procedure and for returning from a remote call were each set at seven units of time. Table 2 gives the minimum, average, maximum, and standard deviation of the speedups in evaluating the ten lookahead trees.

q	f	k	min	avg	max	std
1	2	2	1.21	1.57	2.11	0.28
1	3	3	1.25	2.04	3.27	0.57
2	2	4	1.71	2.37	3.33	0.42
2	3	9	2.13	3.55	6.14	1.00
3	2	8	1.58	3.12	4.58	0.80
3	3	27	1.95	5.31	7.95	1.66

Table 2. Simulated speedup.

Since most game-playing programs must make their move within a certain time limit, any speedup in tree search ability will generally be used to search a deeper lookahead tree. If we have an unlimited supply of processors to form into a binary tree, we can obtain an unlimited speedup only if the search is not limited in time. Otherwise we cannot, because we would eventually violate our premise that the lookahead tree is at least as deep as the processor tree. A new layer on the processor tree does not buy another full ply in the lookahead tree. For example, several speedups of 1.5 would be needed to search a 6-times larger chess lookahead tree, or about one additional ply. The depth of the processor tree would grow faster than the depth of the tree it searches and eventually would catch up. The only way to avoid this limit is to increase the fan-out of the processor tree. If the fan-out is high enough that no successor need ever be queued for evaluation by a slave, then the size of the maximum lookahead tree that can be evaluated within the time limit is limited only by the time required for calls on interior $\alpha\beta$ to propagate from the root to the leaves. Long before this limitation is reached, we would run out of silicon for making the processors.

[†] UNIX is a Trademark of Bell Laboratories.

7. ANALYSIS OF SPEEDUP

We now turn to a formal analysis of the speedup that can be gained in searching large lookahead trees as the number of available processors grows without bound. For this purpose we introduce *Palphabeta*, a simplified version of the tree-splitting algorithm. This algorithm is in general less efficient than the version already discussed, but is more amenable to analysis. Much of the analysis in this section is a "parallelization" of results of Knuth and Moore.² Indeed, when $q = 0$, Theorem 1 and Corollary 1 reduce to their results.

As before, the processors will be arranged in a uniform tree. Let $f \geq 1$ be the fan-out of the processor tree (uniform for all interior nodes), and let $q \geq 1$ be its depth (uniform for all leaf nodes). Let $q + s$ be the depth of the lookahead tree, where $s \geq 1$. We assume that the lookahead tree has a uniform degree and that this degree, df , is a multiple of f , where d is ≥ 2 . Here is *Palphabeta*:

```

1 function Palphabeta(p:position;  $\alpha, \beta$ :integer): integer;
2 var i : integer;
3 function g : integer;
4 begin
5   determine the successors  $p_1, \dots, p_{df}$ ;
6   if depth( $p_1$ ) < q then
7     g := Palphabeta
8   else g := alphabeta;
9   for i := 1 to d do
10    begin
11       $\alpha := \max(\alpha, \max_{(i-1)f < j \leq if} -g(p_j, -\beta, -\alpha))$ ;
12      if  $\alpha \geq \beta$  then return( $\alpha$ );
13    end;
14  return( $\alpha$ );
15 end;
```

The f calls to function g in line 11 are intended to occur in parallel, activating functions existing on each of the f slaves. Serial α - β search is activated on leaf slaves; *Palphabeta* is activated on all others. Unlike the tree-splitting algorithm, *Palphabeta* waits until all slaves finish before assigning additional tasks. However, the two algorithms behave identically when searching either a best-first or worst-first ordered "theoretical" tree of uniform degree and depth. When we restrict ourselves to one of these lookahead trees, we can make conclusions about the behavior of the tree-splitting algorithm by studying *Palphabeta*.

7.1. Worst-first ordering

α - β search produces no cutoffs if, whenever the call $\text{alphabeta}(p, \alpha, \beta)$ is made, the following relation holds among the successors $p_1, \dots, p_d$:

$$\alpha < -\text{negamax}(p_1) < \dots < -\text{negamax}(p_d) < \beta.$$

We call this ordering *worst first*. If no cutoffs occur, it is easy to calculate the time necessary for *Palphabeta* to finish. Assume that a processor can generate f successors, send messages to all of its f slaves and receive replies in time ρ . (This figure counts message overhead time but does not include computation time at the slaves.) Assume also that the serial α - β algorithm takes time n to search a lookahead tree with n terminal positions. Let a_n be the time necessary for a processor at distance n from the leaves to evaluate its assigned position. A leaf processor executes the serial algorithm to depth s . Thus we have $a_0 = (df)^s$. An interior processor gives d batches of assignments to its slaves, and each batch takes time ρ plus the time for the slave processor to complete its calculation. Thus we have $a_{n+1} = d \cdot (\rho + a_n)$. The solution to this recurrence relation is

$$a_q = \rho \frac{d^{q+1} - d}{d-1} + d^{q+s} f^s,$$

which is the total time for Palphabeta to complete. Since the time for the serial algorithm to examine the same tree is $(df)^{q+s}$, the speedup for large s is f^q . There are

$$\frac{f^{q+1} - 1}{f - 1}$$

processors, roughly f^q , so when no pruning occurs the parallel algorithm yields speedup that is roughly equal to the number of processors used.

7.2. Best-first ordering

We will now investigate what happens when the lookahead tree is ordered best-first.

Definition. We will use the Dewey decimal system to name nodes in both processor trees and lookahead trees. The root is named by the null string. The j successors of a node whose name is $a_1 \dots a_k$ are named by $a_1 \dots a_k 1$ through $a_1 \dots a_k j$.

Definition. We say that the successors of a position $a_1 \dots a_n$ are in *best-first order* if

$$\text{negamax}(a_1 \dots a_n) = -\text{negamax}(a_1 \dots a_n 1).$$

Definition. We say a position $a_1 \dots a_n$ in the lookahead tree is (q, f) -critical if a_i is (q, f) -restricted for all even values of i or for all odd values of i . An entry a_i is (q, f) -restricted if

$$1 \leq i \leq q \text{ and } 1 \leq a_i \leq f$$

$$\text{or } q < i \text{ and } a_i = 1.$$

Theorem 1: Consider a lookahead tree for which the value of the root position is not $\pm \infty$ and for which the successors of every position are in best-first order. The parallel α - β procedure Palphabeta examines exactly the (q, f) -critical positions of this lookahead tree.

Proof. We will call a (q, f) -critical position $a_1 \dots a_n$ a *type 1* position if all the a_i are (q, f) -restricted; it is of *type 2* if a_i is its first entry not (q, f) -restricted and $n-i$ is even; otherwise (that is, when $n-i$ is odd), it is of *type 3*. Type 3 nodes have a_n (q, f) -restricted. The following statements can be established by induction on the depth of the position p . (Text in brackets refers to positions of depth $< q$.)

(1) A type 1 position p is examined by calling [P]alphabeta($p, +\infty, -\infty$). If it is not terminal, its successor position[s] $p_1[, p_2, \dots, p_f]$ is [are] of type 1, and $F(p) = -F(p_1) \neq \pm \infty$. This [These] successor position[s] is [are] examined by calling [P]alphabeta($p_1, -\infty, +\infty$). The other successor positions $p_2, \dots, p_{df}$ [$p_{f+1}, \dots, p_{df}$] are of type 2, and are all examined by calling [P]alphabeta($p_1, -\infty, F(p_1)$).

(2) A type 2 position p is examined by calling [P]alphabeta($p, -\infty, \beta$), where $-\infty < \beta \leq F(p)$. If it is not terminal, its successor[s] $p_1[, p_2, \dots, p_f]$ is [are] of type 3, and $F(p) = -F(p_1)$. This [These] successor position[s] is [are] examined by calling [P]alphabeta($p_1, -\beta, +\infty$). Since $F(p) = -F(p_1) \geq \beta$, cutoff occurs, and [P]alphabeta does not examine the other successors $p_2, \dots, p_{df}$ [$p_{f+1}, \dots, p_{df}$].

(3) A type 3 position p is examined by calling [P]alphabeta($p, \alpha, +\infty$) where $F(p) \leq \alpha < +\infty$. If it is not terminal, each of its successors p_i is of type 2, and they are all examined by calling [P]alphabeta($p_i, -\infty, -\alpha$). All of these searches fail high.

It follows by induction on the depth of p that the (q, f) -critical positions, and no others, are examined. □

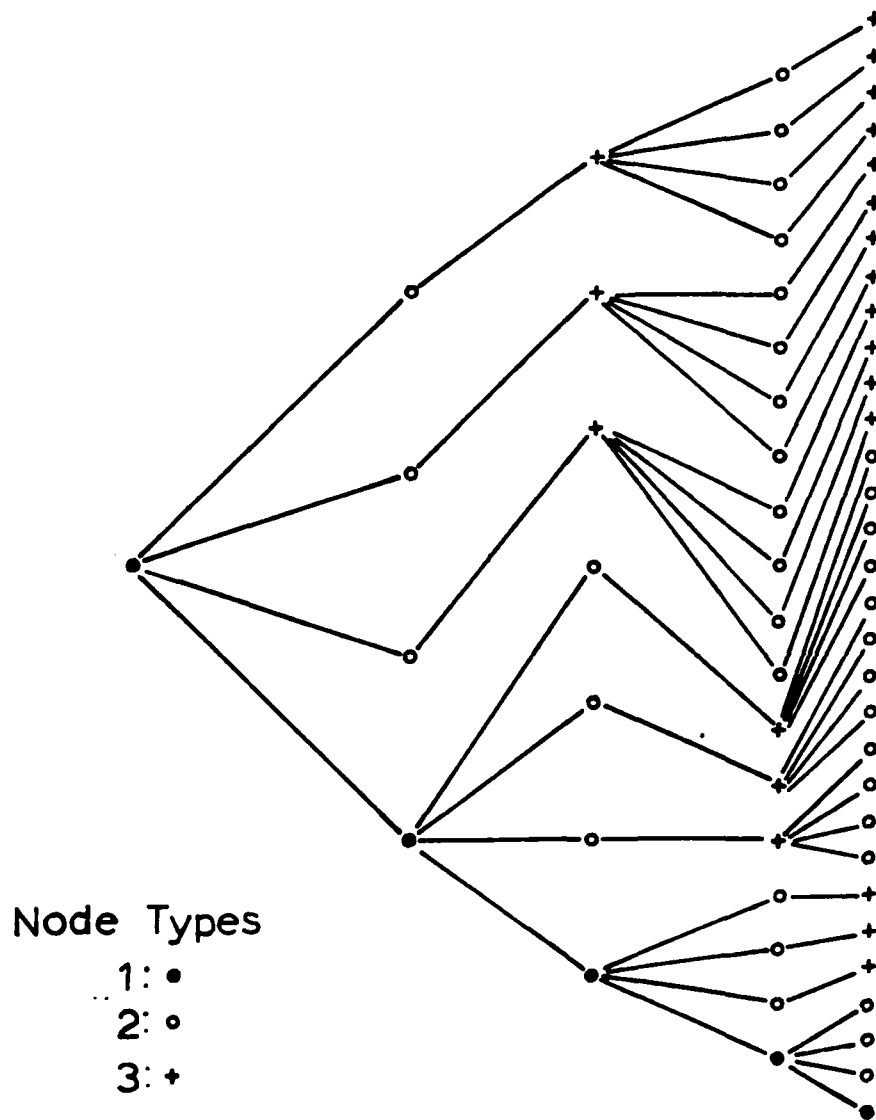


Figure 1. Lookahead tree examined by alphabet.

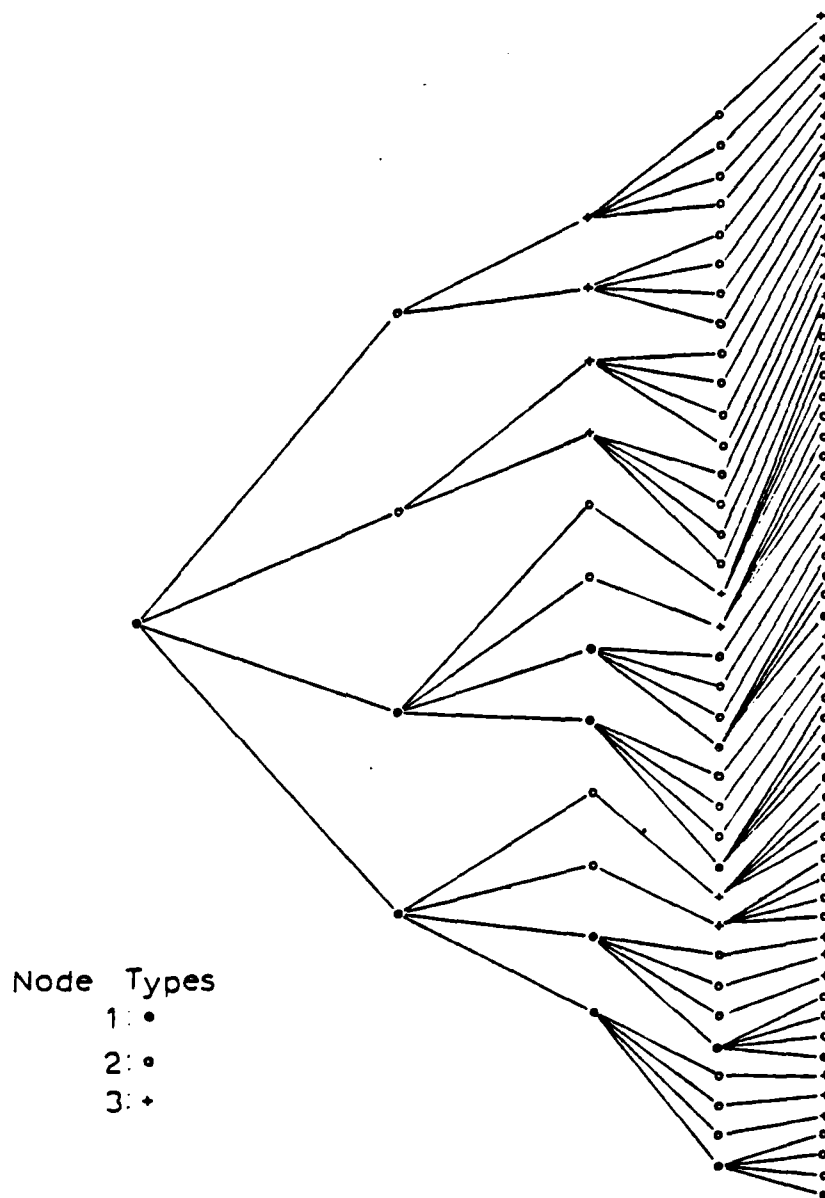


Figure 2. Lookahead tree examined by Palphabeta.

Figure 1 shows the best-first lookahead tree of degree four and depth four that is examined by alphabeta. Figure 2 shows the best-first lookahead tree of degree four and depth four that is examined by Palphabeta running on a processor tree of fanout two and depth two.

Corollary 1. If every position on levels 0, 1, ..., $q+s-1$ of a lookahead tree of depth $q+s$ satisfying the conditions of Theorem 1 has exactly df successors, for d some fixed constant, and for f the constant appearing in Palphabeta, then the parallel procedure Palphabeta (along with alphabeta, which it calls), running on a processor tree of fan-out f and height q , examines exactly

$$f^{(q/2)}(df)^{(q+s)/2} + f^{(q/2)}(df)^{(q+s)/2} - f^q$$

terminal positions.

Proof. There are $f^{(q/2)}(df)^{(q+s)/2}$ sequences $a_1 \cdots a_{q+s}$, with $1 \leq a_i \leq df$ for all i , such that a_i is (q,f) -restricted for all even values of i . There are $f^{(q/2)}(df)^{(q+s)/2}$ such sequences with a_i (q,f) -restricted for all odd values of i . We subtract f^q for the sequences $\{1, \dots, f\}^q$ that we counted twice. □

Palphabeta reduces to alphabeta when $q=0$. Thus Corollary 1 tells us that alphabeta searches

$$d^{(s/2)} + d^{(s/2)} - 1$$

terminal nodes when searching a tree of height s and degree d .

Lemma 1. Given positive constants a, b, c, d , and ρ , the relations

$$\begin{aligned} a_0 &= a; & a_{n+1} &= \rho d + a_n + (d-1)b_n; \\ b_0 &= b; & b_{n+1} &= \rho + c_n; \\ c_0 &= c; & c_{n+1} &= d(\rho + b_n). \end{aligned}$$

are satisfied by the sequences

$$\begin{aligned} a_n &= \begin{cases} (n \text{ even:}) & a + h(n)[d(3\rho+b+c)+\rho-b-c]-n\rho, \\ (n \text{ odd:}) & a + h(n-1)[d(3\rho+b+c)+\rho-b-c]-n\rho + d^{(n-1)/2}(d(\rho+b)+\rho-b); \end{cases} \\ b_n &= \begin{cases} (n \text{ even:}) & \rho + 2\rho g(n) + (\rho+b)d^{n/2}, \\ (n \text{ odd:}) & \rho + 2\rho g(n+1) + cd^{(n-1)/2}; \end{cases} \\ c_n &= \begin{cases} (n \text{ even:}) & 2\rho g(n+2) + cd^{n/2}, \\ (n \text{ odd:}) & 2\rho g(n+1) + (\rho+b)d^{(n+1)/2}; \end{cases} \end{aligned}$$

where the function g is defined by

$$g(n) = \frac{d^{n/2}-d}{d-1},$$

and the function h is defined by

$$h(n) = \frac{d^{n/2}-1}{d-1}.$$

Proof. straightforward algebra. □

Theorem 2: Under the conditions of Corollary 1, and assuming also that (1) serial α - β search is performed in time equal to the number of leaves visited, and (2) in ρ units of time, a processor can generate f successors of a position, send a message to each of its f slaves, and receive the f replies, the total time for Palphabeta to complete is

$$(q \text{ even:}) (df)^{\lfloor s/2 \rfloor} + (df)^{\lfloor s/2 \rfloor} - 1 \\ + h(q)[d(3\rho + (df)^{\lfloor s/2 \rfloor} + (df)^{\lfloor s/2 \rfloor}) + \rho - (df)^{\lfloor s/2 \rfloor} - (df)^{\lfloor s/2 \rfloor}] - \rho q,$$

$$(q \text{ odd:}) (df)^{\lfloor s/2 \rfloor} + (df)^{\lfloor s/2 \rfloor} - 1 \\ + h(q-1)[d(3\rho + (df)^{\lfloor s/2 \rfloor} + (df)^{\lfloor s/2 \rfloor}) + \rho - (df)^{\lfloor s/2 \rfloor} - (df)^{\lfloor s/2 \rfloor}] - \rho q \\ + d^{(q-1)/2}[d(\rho + (df)^{\lfloor s/2 \rfloor}) + \rho - (df)^{\lfloor s/2 \rfloor}].$$

Proof: Let a_n , b_n , and c_n represent the time required for a processor at distance n from the leaves of the processor tree to search type 1, 2, and 3 positions, respectively. A leaf processor searching a type 1 position is actually performing the serial algorithm on a tree of height s and degree df . Hence by Corollary 1,

$$a_0 = (df)^{\lfloor s/2 \rfloor} + (df)^{\lfloor s/2 \rfloor} - 1.$$

Counting arguments similar to those in Corollary 1 give us

$$b_0 = (df)^{\lfloor s/2 \rfloor}$$

and

$$c_0 = (df)^{\lfloor s/2 \rfloor}.$$

In order to evaluate a type-1 position, an interior processor at height $n+1$ in the processor tree orders its slaves to evaluate d batches of successors. The first batch consists of type 1 positions, and the remaining $d-1$ batches consist of type 2 positions. Hence

$$a_{n+1} = \rho d + a_n + (d-1)b_n.$$

Similar arguments give us

$$b_{n+1} = \rho + c_n$$

and

$$c_{n+1} = d(\rho + b_n).$$

By substituting the constant expressions for a_0 , b_0 , and c_0 to find a_q by the formulas given by Lemma 1, we obtain the desired formula. $\square$

Under conditions of best-first search, the parallel α - β algorithm gives order of $k^{1/2}$ speedup with k processors for searching large lookahead trees. The next theorem formalizes this result:

Theorem 3: Suppose that Palphabeta runs on a processor tree of depth $q \geq 1$ and fan-out $f > 1$. Suppose that the lookahead tree to be searched is arranged in best-first order and is of degree df and depth $q+s$, where $d \geq 1$. Denote by R the time for alphabeta to search this tree, and by P the time for Palphabeta to search the tree. Then

$$\lim_{s \rightarrow \infty} R/P = f^{q/2}.$$

Proof: The time for the serial algorithm is

$$(df)^{\lfloor (s+q)/2 \rfloor} + (df)^{\lfloor (s+q)/2 \rfloor} - 1,$$

from Corollary 1. If we divide this quantity by the expression given by Theorem 2 for P , and take the limit as s goes to ∞ , we obtain the desired result. $\square$

7.3. Random Order

We have calculated the finishing time of the tree-splitting algorithm for both best-first and worst-first ordering of terminal positions. Another ordering, called *random order*, assumes that terminal values are independent, identically distributed random variables. Restated, this assumption says that no two terminal values are equal, and that any one of the $n!$ orderings of the terminal values is as likely as any other.

We have partially analyzed a weaker form of Palphabeta, called Pbound, under the assumption of random order. This analysis derives the expected number of terminal positions visited by Pbound. Unfortunately, it does not yield the finishing time, since Pbound sometimes requires processors to be idle. The interested reader is referred to the technical report.⁷

8. DISCUSSION

The improvement that alphabeta search shows over negamax search is due to the cutoffs it achieves. Parallel execution tends to lose some of that advantage, since subtrees that the serial algorithm would avoid are searched before information is available to cut them off. This situation is most extreme if the lookahead tree is ordered best-first; in this case the serial algorithm enjoys the most cutoffs. However, our analysis shows that even in this case, order of $k^{1/2}$ speedup can still be expected. At the other extreme, if the lookahead tree is ordered worst-first, then no cutoffs are found in either the serial or the parallel algorithm. In this case, the parallel algorithm performs no wasted work, and speedup is order of k .

We can now compare the measurements presented in Section 5 with these theoretical bounds. Table 3 compares theoretically-predicted speedups with measured speedups for processor trees of height one and two, and of fan-out two and three.

q	f	k	theoretically-predicted		measured	
			worst-first	best-first	Arachne	simulation
1	2	2	2	1.41	1.81	1.57
1	3	3	3	1.73	2.34	2.04
2	2	4	4	2.00		2.37
2	3	9	9	3.00		3.55
3	2	8	8	2.83		3.12
3	3	27	27	5.20		5.31

Table 3: Speedup

In checkers, certain simplifying assumptions used for the analysis are not true. The lookahead tree is neither regular nor ordered best- (nor worst-) first. Further, the degree of each interior node is not a fixed multiple of f . Therefore, slave processors do not finish in unison. Nonetheless, our implementation results with checkers display speedups that lie between the two analytically derived extremes. These limited results show that the formal analyses are not unreasonable.

9. CONCLUSIONS

The α - β algorithm is central to many game-playing programs. Attempts to speed up this algorithm have usually taken the form of care to order moves in a good approximation to best-first order and special hardware for static evaluation and move generation.

This paper investigates another line of attack: decomposition of α - β search for parallel execution on a multicomputer. The tree-splitting decomposition investigated here assigns

different nodes of the lookahead tree to processors in a processor tree. The penalties for this sort of decomposition are twofold: Communication costs are introduced, and some work is performed that the serial algorithm avoids. Our implementation measurements and formal analysis show that these penalties, although present, do not prevent decomposition from achieving arbitrarily high speedup. Although we do not reach k -fold speedup for k processors, we expect to achieve at least order of $k^{1/2}$ -fold speedup. The loss of efficiency is due almost entirely to lost cutoffs; communication overhead is insignificant.

There are other promising decompositions of α - β for parallel execution; in particular, the "mandatory-work-first" decomposition of Akl *et al*⁴ suggests other dynamic allocations of unfinished work to processors that may result in even greater speedup.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the help and ideas offered by Karl Anderson, Sharon Lawless, Will Leland, Marvin Solomon, and Larry Travis.

References

1. H. J. Berliner, "A chronology of computer chess and its literature," *Artificial Intelligence* 10 pp. 201-214 (April 1978).
2. D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence* 6(4) pp. 293-326 (Winter 1975).
3. G. M. Baudet, *The Design and Analysis of Algorithms for Asynchronous Multiprocessors*, Department of Computer Science, Carnegie-Mellon University (April 1978).
4. S. G. Akl, D. T. Barnard, and R. J. Doran, "Simulation and analysis in deriving time and storage requirements for a parallel alpha-beta algorithm," *Proc. 1980 International Conference on Parallel Processing*, pp. 231-234 (August 1980).
5. P. Brinch Hansen, "Distributed processes: A concurrent programming concept," *CACM* 21(11) pp. 934-941 (November 1978).
6. M. H. Solomon and R. A. Finkel, "The Roscoe distributed operating system," *Proc. 7th Symposium on Operating Systems Principles*, pp. 108-114 (December 1979).
7. J. P. Fishburn and R. A. Finkel, "Parallel Alpha-Beta Search on Arachne," Technical Report 394, University of Wisconsin--Madison Computer Sciences (July 1980).

APPENDIX F

**The Arachne Kernel
Version 1.2**

*Raphael Finkel
Marvin Solomon*

Technical Report 380
Computer Sciences Department
University of Wisconsin--Madison
April 1980

THE ARACHNE KERNEL *

Version 1.2
April 1980

Raphael Finkel
Marvin Solomon

Technical Report 380

Abstract

Arachne is a multi-computer operating system running on a network of LSI-11 computers at the University of Wisconsin. This document describes the implementation of the Arachne kernel at the level of detail necessary for a programmer who intends to add a module or modify the existing code. Companion reports describe the purposes and concepts underlying the Arachne project, present the implementation details of the utility processes, and display Arachne from the point of view of the user program.

* This research was supported in part by the United States Army under contract #DAAG29-75-C-0024. We have been forced to change the name of the Roscoe distributed operating system, since Roscoe is a registered trademark of Applied Data Research, Incorporated. The new name we have chosen is Arachne; the operating system and research continue unchanged.

TABLE OF CONTENTS

1.	INTRODUCTION.....	1
1.1	Revisions.....	2
2.	PREPARING THE KERNEL.....	4
2.1	Source Files.....	4
2.2	Compilation and linking.....	6
2.3	Loading and Starting.....	6
3.	FUNDAMENTAL MODULES.....	9
3.1	Initialization.....	9
3.2	Low level interrupt handling.....	11
3.3	Debugging Aids.....	12
3.4	I/O.....	15
3.5	Free storage management.....	16
3.6	Locks.....	17
3.7	Service calls.....	18
3.8	The clock.....	21
4.	PROCESS MANAGEMENT.....	24
4.1	Scheduler data.....	24
4.2	Initiating processes.....	25
4.3	Process switching.....	27
4.4	Process switching in Elmer.....	29
4.5	Treatment of schedcall.....	29
4.6	Core images.....	34
4.7	Timing.....	38
5.	MESSAGES.....	40
5.1	Central message handling.....	40
5.2	Inter-machine Messages.....	44
5.3	Links.....	47
5.4	User messages.....	51
5.5	Asynchronous Message Receipt.....	54
6.	INTERRUPTS AND EXCEPTIONS.....	56
6.1	Interrupts.....	57
6.2	Exceptions.....	58
7.	ACKNOWLEDGEMENTS.....	60
8.	REFERENCES.....	60

THE ARACHNE KERNEL

1. INTRODUCTION

The Arachne project at the University of Wisconsin is implementing a distributed operating system for several co-operating LSI-11 microcomputers [1,2,3]. Documentation for programmers writing code to be run under Arachne can be found in a companion report [4,5,6]. Arachne is in a state of flux; the details of the kernel are likely to have changed since the time this report was written.

The operating system is divided into the kernel and the utility processes. The kernel manages storage, process creation, deletion, and scheduling, and messages. Utility processes provide terminal handling, file management, interactive command line interpretation, and resource allocation.

This report describes the Arachne kernel at the level of detail necessary for systems programmers who might be modifying or inserting code. Utility programs are documented in a companion report [7] and further described elsewhere [8].

All code for Arachne is written in the C language [9] or in the Unix [10] PDP-11 assembler language. The C compiler has been

modified to provide stack-limit checking at the entry to each procedure, since the LSI-11 does not have any hardware stack limit check or memory management. The Arachne kernel is compiled and linked on Unix on a PDP-11/40 and is then sent to the various LSI-11 machines through DR-11C (sixteen-bit parallel) lines. Details of the linking and sending procedures are given below.

User programs that run under Arachne are written either in C or in Elmer, a new language developed by the Arachne project. Elmer has the advantage that its code is completely relocatable and the language is strongly typed. Various parts of the kernel, particularly those dealing with core images (Section 4.6) treat C and Elmer programs differently.

1.1 Revisions

Version 1.1 of Arachne differs in several significant ways from Version 1.0 [4,11]. First, Elmer has been introduced. Interfaces for this language have caused changes in the scheduler and service call dispatcher. Second, the driver for the DRV-11 lines that connect the machines has been completely revised, with a large resulting gain in speed. Third, link tables for processes can now expand so that the occasional process that needs a large table can get one. Fourth, certain locations in low core are used to store certain state variables to assist in debugging and tuning. Fifth, process identifiers are now unique even across restarts of Arachne. Sixth, messages now include length information, so the various copying steps can be sped up

In the case of short messages.

Version 1.2 differs from Version 1.1 [5] in the following ways: First, the checksum is periodically checked on the kernel code. Second, the message receipt structure "usmsg" has been abolished, and "urmsg" no longer has a field to hold the message that is being received. Instead, the sending and receiving procedures now take an explicit argument pointing to the message string. This modification enables Elmer programs to use standard send and receive. Third, the service call convention for Elmer has been modified and the special cases for Elmer in the scheduler have been largely removed. Fourth, if a bit in the global debugging location is set, checksums are verified for each process before it is executed. Fifth, a new link restriction, MAYERROR, allows the link holder to send an error message, receipt of which raises an exception. Sixth, the "userdie" service call now takes an argument that becomes the body of any resulting DESTROYED messages. Kernel-induced termination of processes also provides information in the body of the DESTROYED messages. Seventh, exceptions may be raised during the execution of service calls. All USERERRORs raise exceptions. Uncaught exceptions terminate the process that caused them. Processes may catch exceptions. Eighth, a new service call "usercatch" allows a process to establish a routine that will catch messages asynchronously. Ninth, the clock routines have been completely rewritten to fix an 88 time loss and to simplify the algorithm. Wakeups are now accurate only to the nearest second. Tenth, facilities have been added for gathering statistics on the CPU usage of processes.

Finally, we have been forced to change the name of the Roscoe distributed operating system, since Roscoe is a registered trademark of Applied Data Research, Incorporated. The new name we have chosen is Arachne; the operating system and research continue unchanged.

2. PREPARING THE KERNEL

2.1 Source Files

All files for Arachne reside in the directory /usr/network/roscoe. Each source file has a name and an extension, for example, "clock.h" has the name "clock" and the extension "h". The extensions follow this pattern:

<u>extension</u>	<u>purpose</u>
u	source file in C, should be compiled with the stack-limit-checking compiler.
c	source file in C, should be compiled with the normal compiler.
s	source file in assembler.
h	header file for C, defines global data structures meant to be included in several modules.
o	object file produced from corresponding u, c, or s file.
lda	object file in LSI-11 absolute loader format.

The source files for utility processes are in the subdirectory user/util. These files are all of the "u" and "h" variety. The Arachne loader, although part of the kernel, is also in this directory, since it follows user, not kernel, naming conventions. The object files for all user processes, including utility

processes, are stored in the subdirectory "user". Many library routines are available in the library "libr.a" in that subdirectory. The source for these routines is in the subdirectory "library". Some Elmer library routines are in the subdirectory "elmer".

The kernel source code is mostly found in thirteen "u" files. In addition, there are three "c" files, 2 "g" files, the loader (a "u" file in the "user" subdirectory), and two empty files that define entry points for the loader. Each file begins with an extensive comment naming the procedures, data, and structures that are imported into, exported from, and local to that module. Many modules use "h" header files to communicate exported data and structures. The cross-reference file "cref" contains a complete listing of all procedures that are defined in one module and used in other modules. This file lists each procedure name, the defining file, and the names of all referencing files. It is also easy to find all uses of a particular function, say "foo", by using the "grep" program:

```
grep foo *.u *.c *.s
```

This program searches for all mentions of "foo" in all source files.

2.2 Compilation and linking

The kernel can be compiled and linked by invoking "lprep". This command file calls "compile" on each source file for the kernel. The program "compile" compares the date on the source file (whether its extension is "u", "c", or "s") with the date on its object file (extension "o"). If the former has been changed since the latter was prepared, the appropriate compiler/assembler is invoked to create a new object file. The linker is then invoked to combine the various object files. Finally, "lprep" invokes "mkida" to convert the Unix object file format into absolute loader format, producing "arachnew.lda". After this version has been checked out, it may be moved into "arachne.lda", which is the current working version of Arachne.

2.3 Loading and Starting

To load a new version of Arachne into a cleared LSI-11, use the Unix program "com1". This program acts as the console terminal to the LSI-11 and can send whole files either on the console line or through the fast word-parallel (DR-11C) line. By default, "com1" uses the fast line to machine 0. To set it to, say, machine 1, type "<control-T>1".

In the cleared state, the LSI-11 executes microcode that implements ODT (octal debugging technique) [12]. To enter ODT at any time, send a <break> to the console. The program "com1" sends a break when the user types "<control-T>B". ODT prompts

for input with "q". To invoke the hardware absolute loader, follow this procedure:

who	what	meaning
user	<ctr-T>B	Interrupt LSI-11
LSI	q	prompt
user	173000G	start a diagnostic program at 173000.
LSI	\$	prompt
user	AL<cr>	starts the absolute loader
user	<ctr-T>S	send along slow line
com1	file =	prompt for file name
user	drload.lda<cr>	name of software loader
com1	sending	
LSI	Loading	software loading starting

If the PDP-11/40 is not connected to the LSI-11 console, but a terminal is, then the following program should be entered through the terminal:

location	contents
1000	005737
	167730
	002375
	013721
	167734
	077006
	000000
R0	000273
R1	157000

Once these values are loaded, start the LSI-11 at location 1000 (command 1000G), and send the file "drload.abs" across the fast line to the LSI-11. (See below for two ways to send files across the fast line.)

Once "drload" is present, it can be started by typing "157000G" to ODT. The message "Loading" should appear immediately. After this message appears, there are two ways to load the kernel:

who	what	meaning
user	<ctr-T>P	send along fast line
com1	file =	prompt for file name
user	arachne.lda<cr>	name of kernel file
com1	sending	
drload	Starting	

The other way is to give the Unix command
cp foo /dev/dr7

where 7 should be replaced by the line number for that machine.

Several core locations are intended not to be disturbed between loads; however, the hardware loader can damage these numbers. They are as follows:

location	meaning	suggested value
157700	machine no.	0 to 4
157702	print delay	0 for fast terminal (crt) 2000 for com1 at 1200 baud 3000 for com1 at 300 baud anything unusual
157704	nextid	

The variable "nextid" is used to prevent processes started under the new version of Arachne from having the same id as those under the old version. Only the lower eight bits of "nextid" need to be set.

It is possible to restart Arachne at any time by halting execution and starting at location 700 (ODT terminal command 700G).

3. FUNDAMENTAL MODULES

3.1 Initialization

When Arachne is started, a small amount of code in module "ctrl.s" causes a Unibus reset to clear all interrupt enablings, sets the kernel stack to location 700, and jumps to routine "main" in module "lsi.c". This routine sets the kernel stack limit to 400 (in global location 1000), inhibits interrupts, and invokes "initall", which calls the initialization code for each module.

After all modules are initialized, "initall" prints the name of the operating system, the machine id, and the amount of memory used.

The "main" routine then establishes the periodic checksum calculation by calling "dochkaum", which reschedules itself on the clock (Section 3.8) every 10 seconds. Finally, "main" initializes the kernel job "kernjob" in "kernjob.u". (For details of scheduling, see Section 4.) Then "startscheduler" in module "schedule.u" is called.

When the kernel job starts, it acts as a normal process controlled by the scheduler. It first prompts the terminal for one of three methods of proceeding. The first is to load no programs, but to wait in a kernel-job loop for a message to arrive from a foreign site. The second is to load in a program. If this option is chosen, the kernel job submits a normal user load

request that will prompt the terminal for the file and will read it across the fast line. Methods like those given in Section 2.3 can be used to send the desired program. After the program is loaded, it is allowed to execute, and the kernel job remains in a loop waiting for either a message from a foreign site or for the termination of the loaded program. If the latter occurs, the kernel job again presents the three options. The third option is to load a new resource manager that should be linked into the web of existing resource managers. The kernel will prompt for the configuration desired for this machine (that is, what other utility processes should be resident) and for the machine number of a working Arachne site. The details of starting resource managers are outside the scope of this document; see [7]. The kernel job interface to the resource manager is one of the two instances in which the kernel needs to know details of utility processes. (The other instance is the loader, which needs to know file system protocols. The loader is discussed in Section 4.6.)

Files ctrl.s, lsi.c, kernjob.u

Procedures

main() Calls initall, then starts up kernjob.

initall() Calls the initialization code for each module.

kernjob() Loads the first user program, then waits for messages from remote kernels to load up new programs.

dochkaum() Computes the checksum, complains if an error is found. Sets a clock event that will run it again in 10 seconds.

3.2 Low level interrupt handling

A small module, "lowestirp.s", is in charge of dispatching the various interrupts that may occur. These routines generally save machine registers r0 and r1 and call a C routine in the kernel. When that routine is done, the registers are restored and the interrupt is dismissed. The interrupts dealt with are the clock interrupt, which calls "timesup" in "clock.u" (Section 3.8) and the interrupts from the various driver lines to other machines, which are dispatched to "inhandle" or "outhandle" in "line.u" (Section 5.2). These latter interrupt routines also place an argument on the stack to tell which driver line caused the interrupt. User interrupts (Section 6.1) are caught by "lowus" and "lowul" (and others, as more are allowed) and dispatched to "uinterrupt" in "interrupt.u". Trace traps are caught by "tracetrp", which prints the location of the trapping instruction and allows the user to continue either with trace trap on or off. This routine is seldom used, because it seems to be unreliable.

In addition, "lowestirp.s" provides the routines "pause", used to display a number and halt the machine (for debugging; see Section 3.3) and "psset", to set the processor priority (for locks; see Section 3.6).

Files lowestirp.s

Data

irprout

A table for routing driver line interrupts.

irprsize The length of "irprout", used in initialization of the line handlers.

lowulnt

A table of user interrupt handlers.

Procedures

clktint

Calls "timesup" when the clock ticks.

setflag

During parts of initialization, non-existent memory traps are dispatched here to set a global flag that can be examined.

int psset(newps)

Sets priority to newps (actually, priority left-shifted by 5 bits). Returns old priority.

pause(code)

Displays the code and halts. Can be continued by the ODT command "p".

tracetrp

Prints the current program counter, accepts X to leave trace trap on, anything else to turn it off.

3.3 Debugging Aids

The module "error.c" contains various debugging aids. This module is compiled with the standard C compiler, since stack overflow calls "syserror" in this module.

The routine "pause" in module "lowestirp.s" prints its default argument and then halts. The console ODT command "p" will proceed from the halt.

When the Arachne kernel discovers a situation from which it cannot recover, the macro SYSEERROR (defined in "util.h") calls the routine "syserror" in module "error.c". This routine prints a coded message to the console terminal and then executes a

"pause(-1)". The messages are listed in the file "syserrors"; their text is not stored in the kernel proper in order to save space. It is usually not wise to continue from such an error.

Non-existent memory traps and illegal instruction traps cause a message to be printed from routine "nmerror". The processor interrupt vector is established during "initall" in "lsi.c". If the error is in the kernel, syserror is invoked. Otherwise "userdie("bad trap")" in "schedule.u" is called to terminate the offending process.

Every ten seconds the kernel runs "dochksum" in "lsi.c" to insure that its code space has not been damaged. The new checksum is computed and stored in the variable "checksum" if that variable holds a zero; otherwise, the new checksum is compared with the current one. If there is a discrepancy, a syserror is signalled.

Service routines (Section 3.7) return failure to the calling process by invoking the macro USERERROR (defined in "util.h"), which calls "usererror" in "error.c" and then returns a negative error code. This procedure raises an exception by setting "cureerror" in "schedule.u" to 1 and then prints a debugging message. A list of errors can be found in the file "usererrors".

Exceptions can also be raised by receipt of an error message. If the user has invoked "uerhandle" in "schedule.u", then when the service call during which the exception was raised returns through the wormhole, the user-supplied routine is invoked with the same arguments as the service call. In addition, two extra arguments are placed in front of the old argument list: the

service code and its returned value. The user routine will return to the calling point within the user program. If "uerhandle" has not been invoked, raised exceptions cause a call to "die("exception not caught")".

If bit 929 is set in "debug", then the scheduler will verify the checksum of the text segment of each process right before it is scheduled. Processes whose text segments have been garbled are terminated.

The routine "snap" in module "error.c" checks its argument against the global variable "debug", which is always stored in location 776. If "debug" and the argument to "snap" have any bits in common, then a walkback of routine return addresses is printed to the terminal and "pause" is invoked.

The global location "debug" can also be used as a guard on various diagnostic output while new modules or routines are being tested.

In addition, several important variables are kept in low core to facilitate debugging.

location	name	meaning
776	debug	flags for debugging
769	curusr	process number of current process
756	numkms	number of remaining message buffers
762	curerror	whether an exception has been raised

Several debugging routines that print various structures can be found in the file "debug.u". These routines are not normally included in the kernel but can be edited into the kernel source during debugging. They include "printprocs" to print the status of each process, "printkm" to print a kmesg buffer, "printin" to print a link, "prntclkq" to display the clock queue, "chkwt" to

check the consistency of the message arrival queues, and "free-print" to print a map of free space.

Files

lowstirp.s, error.c, debug.u

Procedures

pause(code)
Print the code on the terminal and halt.
snap(tag)
If location "debug" and "tag" have any bits in common, print a traceback of procedure frames and pause(tag).
nxerror(dummy)
Print message, then either call schedcall or syserror.
Print a trace of the stack and pause.
syserror(code)
Print the code (which can be found in the file "syserrors", then call "pause(-1)".

3.4 I/O

The lowest-level input-output routines that deal with the console terminal are in module "io.c". Since these routines can be called after a stack limit exception, they are compiled by the normal C compiler that does not check for stack overflow. The routine "outchar" sends a character to the console terminal after sitting for a while in a busy loop. The number of iterations of this loop is stored in the global location "delaylen" at 157702 (octal), which can be set in ODT. The delay is to slow down the natural rate of transfer so that if the program "com1" is receiving the console stream, Unix can keep up with the line.

The "printf" routine used by the kernel is in module "lsi.c". It acts much like "printf" in Unix, except it does not have widths in its format specifications. "Printf" is only used for debugging output; terminal I/O is ordinarily handled by the terminal driver utility process (6,7). The module "lsi.c" de-

lines "putchar", which calls "outchar" in "io.c".

Files

io.c, lsi.c

Procedures

outchar(dev,c) char c;
Print one character to the terminal at address dev.
char inchar(dev)
Read one character from the terminal at the address dev.
ttyflush()
Remove any waiting character from the teletype input data register.
printf(msg,arg) char *p_msg;
Emergency and debugging output using putchar().
putchar(c) char c;
Same as outchar(TTY,c)
char getch() Same as inchar(TTY)
Same as inchar(TTY)

Data

delaylen

Location where tty delay is stored, defined in crtl.s

TTY

Address of console terminal registers (0177560)

3.5 Free storage management

Free storage management routines are found in the module "free.u". During Arachne initialization, the routine "freeinit" grabs a large section of memory from the end of the kernel using the "sbrk" routine in "lsi.c". The boundary tag method [13] is used to allocate chunks of storage through the routine "freeget" and to return them through the routine "freerel". The regions that are returned are initialized to contain -2 in every word to catch subsequent initialization errors in the users of the regions. Free storage is used to find room for certain kernel tables that are made once during Arachne initialization and to provide room for core images. This latter space is reclaimed by a reference count technique. (See Section 4.6.)

Files

free.u, lsl.c

Data Structures

int freesize

Length of the free space in words

int *freespace

Start of the freespace. Set by sbrk(freesize) at initialization.

int freeptr

Head of a doubly linked chain of blocks of free storage.

struct { int fsize, *fnext, *fprev; }

Format of the header of a free block.

char *highwater

Highest address allocated by sbrk.

Procedures

freeinit()

Free storage initialization

int *freeget(size)

Returns a block of "size" words of free storage. Returns 0 if the required space is unavailable.

freerel(ptr) int *ptr

Release the block of storage pointed to by "ptr".

char *sbrk(count)

Allocates "count" bytes of storage by extending highwater.

int outrange(addr) char *addr;

Returns TRUE if addr is not in any reasonable user range.

3.6 Locks

Several sensitive queues must not be simultaneously modified by an interrupt-level routine and a standard routine. Access to these queues is controlled by interlocks found in the module "lock.u", which provides routines "wait" and "signal". The "wait" routine makes sure that no lock is active, then places the CPU at high priority by calling "psst" in "lowestirp.s". (The LSI-11 has only two priority levels.) The "signal" routine restores the original priority (which may have been high). No interrupts are serviced between these calls. Four queues are locked in this fashion and will be discussed later: the ready

process queue, free message buffer queue, received message queue, and outgoing message queues. In addition, several routines in "clock.u" and "timing.u" use "wait" and "signal" instead of clearing and resetting the clock interrupt enable bit, since clearing it sometimes causes a bus error during acquisition of the clock interrupt vector.

Files

lock.u and lock.h

Data Structures

int NUMLOCKS

Number of lock types. These locks are defined:

RQLOCK: ready queue lock; used in schedule.u

KMBUFLOCK: available list of message buffers; used in kmess.u

IOLOCK: list of outgoing messages; used in line.u

KMTLOCK: list of received messages; used in kmess.u

CLKLOCK: event queue lock; used in clock.u in lieu of disabling the clock

int curlock

-1 if no lock active, else name of lock.

int lastps

processor status before the current lock was entered.

Procedures

lockinit()

Initialization routine.

wait(lock)

Make sure no lock is in force, then move to high priority.

signal(lock)

Make sure the argument is the lock currently in force, then return to the old priority.

3.7 Service calls

Processes running under Arachne request kernel assistance by invoking kernel service routines. C programs call these service routines by placing arguments on the stack, placing a service code in register r1, and then subroutine-jumping to location

1002.

Elmer programs place the arguments at the start of data space instead. The user's registers r0 and r1 are saved on the user's stack, and the arguments to the service routine are taken from the start of the user's data area and placed on the stack. The service routine proper is called by a subroutine call, so when it returns, "sys" can restore the registers and compute where in the Elmer program to return. The return address is found by adding r0 and r4. (Register r4 points to the start of code space, and r0 is an offset in that space.)

The routine "sys" in the module "ctrl.s" resides at 1002. It has access to the scheduler variable "curtype", which indicates whether the currently running process is of the C or Elmer type. It decodes these calls and invokes the appropriate kernel routine. Each service call is represented by two words in a table, indexed by the service code: The first is the address of the service routine, and the second is a set of flags indicating whether the routine may be called at interrupt level and at normal level, and whether it is a privileged instruction. No use is currently made of the privileged flag. (Interrupt handling is discussed in Section 6.1.) This same routine calls the routine "maydic" in "schedule.u" so that the calling process may be terminated if termination is pending. (See the discussion on the scheduler, Section 4.)

When the service routine is finished, control is returned to the wormhole routine, which checks whether an exception was raised during its execution. If not, execution continues at the

user's point of call. If an exception has been raised and the user has not established an exception handler (via "uerrhandle" in "error.c"), then the user is terminated. If an exception handler has been established, it is called with the stack argumented to appear that the user had called it directly instead of the service call. Its arguments are the returned value from the service call, the service code, and then all the arguments to the service call in order.

During execution of the service routine, the stack belonging to the calling process is used. For this reason, most kernel routines are compiled with the stack-checking version of the C compiler. The service routine must return to the wormhole and it must access its arguments on the stack, but the wormhole itself needs to save two local variables (the service code and the return address to the user), so a copy of the arguments is made before the service call is actually invoked. At the point that the service call is invoked, the stack looks like this:

```
wormhole return address
arg 1 copy
...
arg NMARGS copy
service code
user return address
arg 1
...
arg NMARGS
```

If the user-supplied exception handler is invoked, the stack is set this way:

```

user return address
service routine return value
service code
arg 1
...
arg NARGS

```

Service routines usually have two names: the one that applies in the kernel, and the alias employed by user processes.

The aliases are defined in a companion report [6].

Files
crtl.s

Data Structures
Table

Branch table for dispatching service calls.

Procedures
sys

At location 1002. Dispatches service calls.

3.8 The clock

The module "clock.u" provides a routine "setalarm", which allows an arbitrary routine with up to four arguments to be called some number of seconds in the future. If the caller of "setalarm" specifies a "delay" of n , then the given routine will be called when the clock has interrupted $n+1$ more times. The hardware clock interrupts once per second, so the given routine will be called between n and $n+1$ seconds from the time setalarm is called. The scheduled routine will run at high priority. The routine "setalarm" returns a code that can be used to turn the alarm off before it expires by invoking "turnoff" with that code as an argument.

The clock routines are implemented with a programmable clock, which is set to interrupt at one-second intervals. The

routines "setalarm" and "time" use locks (Section 3.6) to prevent clock interrupts from occurring at critical times. (Clearing the interrupt-enable bit can make the processor halt.)

A queue of events stores the alarms that have been set; each node includes the time, in seconds from the epoch (beginning of 1973, Madison standard time), when the event is to occur.

When the clock interrupts, it calls "timesup" at interrupt level to increment "datereg" and "timeofday", "nexttime" in "timing.u" to increment "timeslot" for statistics gathering (Section 4.7), and to execute all events on the queue whose time has expired. The clock stays in repeat interrupt mode, so Arachne's notion of time is maintained with the accuracy of the clock hardware.

The date, given in seconds since Jan 1, 1973, may be obtained with the service routine "date", which returns a long integer. The service routine "setdate" may be used to change this date. Another service routine, "time", returns a long integer that counts in increments of .0001 seconds. This latter timer cannot be modified. "time" returns the sum of "timeofday" and the elapsed time since the last clock interrupt. This elapsed time is determined from the clock's count register, which decrements by one every ten-thousandth of a second. These services are used both by user programs and outgoing message sending and message reception, which are discussed in Sections 5.2 and 5.4.

Files

clock.h and clock.u

Data Structures

```
int NBRARGS
number of arguments to alarm routine, currently four
struct cqvector {
    int (*cqfunc)(); /* function to call when alarm expires */
    int cqargs[NBRARGS]; /* arguments to that function */
};
char *timeofday
interval timer in seconds
long datereg
seconds since the epoch (beginning of 1973, Central standard time)
int uniquecode
used to distinguish alarm events
struct clknode {
    struct clknode *cnext; long expire; /* time when this event is to occur; in seconds from the epoch */
    int ccode; /* distinguishing code */
    struct cqvector cqaction;
};
struct clknode clkqueue[CLKQSIZE]; /* clkqtop, *clkqfree
Queue of alarms, start of queue, head of free list of queue nodes.
```

Procedures

```
long time()
Returns the current interval timer value (in .0001 seconds), as determined from "timeofday" and the clock count register.
long date()
Returns the current value of "datereg".
setdate(n) long n;
Sets datereg equal to n.
int setalarm(delay,action) struct cqvector *action
Places an alarm on the queue. When it expires, the action will be taken. Returns a code to be used for "turnoff".
turnoff(code)
Remove the alarm that has the given code.
timesup()
Called by a clock interrupt every second to run the pending events.
clockinit()
Initialization of the clock data structures and clock hardware. The clock registers are not modified again after this routine finishes.
```

4. PROCESS MANAGEMENT

Processes are managed by the scheduler and the core image routines. There are two kinds of core images, C and Elmer. C images contain code, initialized data, and space for the uninitialized data. Elmer images contain only code. They are treated slightly differently by the modules that handle processes.

4.1 Scheduler data

The scheduler resides in "schedule.u". This module keeps an array of per-process information, which includes a stack frame pointer, the current status of the process (non-existent, sleeping, ready, running, halted, to be halted, to be terminated), a pointer to the next process on the ready queue (if this process is itself ready), the lowest address allowed for the stack, an index into a reference count table for purposes of storage reclamation, the type of the core image (Elmer or C), the address of the data area (for Elmer processes only), the exception handler address (Section 6.2), asynchronous message handler address (Section 5.5), arguments and active channels for asynchronous message receipt, and timing information (Section 4.7).

Each process is identified by two numbers: an index into the process table (process number) and a network-wide unique identifier (process id). The module "schedule.u" provides routines "getid" and "getno" for conversion between the two. The high byte of the process id is the machine id; the low byte is a se-

quence number. The variable "nextid" is saved across instantiations of Arachne so that process identifiers are not reused soon after a Arachne machine goes down and then comes up and rejoins the network. When a new process is to be assigned its process id, the low byte of nextid is incremented so long as it is equal to any extant process id. After nextid is assigned to the new process, the low byte of nextid is incremented. Process ids are used as message destinations, since it is wrong to direct a message to a new process that happens to have the same process number as the desired but terminated process.

4.2 Initiating Processes

A process is started by a call to "initiate", which takes a core image, an argument, an owner id, and, for Elmer, a file descriptor. A new stack is created and initialized so when the process starts, it will appear as if it has been called with the given argument. The stack is also prepared with a return address that points to "fallthrough", so if the process returns, "fallthrough" can perform a "schedcall(DIE)" and properly terminate it. For an Elmer image, a smaller stack is allocated, since Elmer allocates all local variables in its data area rather than on the stack. In addition, room for the Elmer data area is created and initialized by calling "uload" in module "uloader".

In either case, the reference count on the core image is incremented to indicate that another process is running in it. A check is made to insure that the owner of the core image is performing the initiate. Finally, the process is awakened by a call

to "awaken".

The routine "awaken" causes a process whose status is "sleeping" to be placed on the ready queue with status "ready". It does nothing if the process was not sleeping. (For this reason, "initiate" first sets the status to "sleeping" and then calls "awaken".) The usual reason that processes enter the sleeping state is to await a message. It never hurts to spuriously awaken a process, since it will check to see if the right message has arrived and will return to the sleeping state if it hasn't. (See Section 5.1 on messages.)

Scheduling is round-robin non-pre-emptive. The scheduler itself remains in a loop in routine "scheduler". Once a process has been chosen for execution, the scheduler loop verifies the checksum in its text area (if the 020 bit is on in global location "debug"), then it calls "strust" (in "resume.s"), which switches to the process stack and transfers control to that process. The stack limit for the new process is placed in location 1000 (octal) so that procedure entry code can check for stack limit violations. (Elmer programs do not use the stack, but the kernel routines they invoke use the small stack in the Elmer program.) Interrupts are prevented during all stack switching, when the stack limit implied by location 1000 is not consistent with the stack pointer in hardware register r6. (Since interrupt-level routines also use stack-limit checking, they would likely signal a stack overflow.) Processes are run at low priority.

4.3 Process switching

The details of process switching depend on the standard C subroutine linkage conventions. Normally, each activation of a procedure has a corresponding stack frame. More recent activations correspond to stack frames at lower addresses. A frame is identified by an address (called the frame pointer) of one of its fields. The word addressed by the frame pointer contains the frame pointer for the next older (higher addressed) frame. The following word (next higher address) contains the return address, and subsequent words contain the actual parameters, the first actual parameter being at the lowest address. The three words preceding the frame pointer word are used to save registers r2, r3, and r4. Subsequent locations are used for local variables and temporary storage. Register r5 always contains the current frame pointer:

```

r6 --> temporary storage
      local variables
r5 --> saved registers
      previous frame pointer
      return address
      first actual parameter
      second actual parameter

```

(Lower addresses are towards the top of the diagram.)

To call procedure "foo", say, the calling program pushes the actual parameters onto the stack and then executes a "jar pc,foo" or a "jar pc,\$foo" instruction, thus pushing the return address onto the stack. If the program "foo" was compiled without stack limit checking, it first executes "jar r5,csv", which pushes the previous frame pointer. (The routines "csv", "ncsv" and "cret"

are in the C library.) The routine "csv" then saves r2, r3, and r4, sets r5, and returns to "foo". The code of "foo" then decrements r6 to make room for local variables. The stack pointer r6 ends up pointing one word beyond (below) the last local variable.

The stack-limit-checking version of "foo" is similar, but instead of calling "csv" and then making room for local variables, it places the negative of the number of bytes of local variables into r1 and calls "ncsv", which checks the limit (in location 1000) to see that the space for local variables can be granted with room to spare and then decrements the stack pointer. Return from a function is accomplished in either version by a jump to "cret", which restores the registers r2, r3, r4, and r5 to their values prior to the call, sets r6 to point to the return address on the stack, and executes an "rts pc" instruction. Actual parameters are cleared from the stack by the calling program. (All parameters are passed by value.) Just before returning from a procedure, "cret" sets r6 to point to the word following the word addressed by r5 during the procedure, that is, the return address. Hence, the entire state of a process can be saved by storing r5, provided the process is just about to return from a procedure.

A process that wishes to relinquish control calls "schedcall". The code in "schedcall" stores the current r5 in the stack frame pointer field for the current process, replaces it by the saved r5 from the scheduler "process", and returns. Similarly, "strtusr" resumes a process by restoring the saved r5 value and doing an ordinary return. To the process, it looks as if the

call to "schedcall" returns immediately.

4.4 Process switching in Elmer

Elmer processes expect that r5 points to the data area (which includes both initialized and uninitialized data), and the first word of the data area points to the start of the code area. When an Elmer process makes a service call, it places an argument count and the arguments themselves starting at the second word of its data space. The return address is in r3, represented as an offset from the beginning of the code space. The service call is handled by "sys" in "crtl.s", which places the arguments on the stack, so that the scheduler can treat Elmer and C programs alike.

4.5 Treatment of schedcall

The argument to "schedcall", one of DIE, CONTINUE, and SLEEP, is passed back to the scheduler as a result of the invocation of "strctust". In the case of DIE, "killoff" is invoked and another process is chosen at the start of the scheduler loop. In the case of CONTINUE, the process remains ready but is placed at the end of the ready queue. In the case of SLEEP, the process is placed in state "sleeping". This alternative is used by service routines that need to wait an unspecified amount of time and are willing to allow other processes to execute in the meantime. Forms of schedcall(DIE) and schedcall(CONTINUE) are provided for processes in the service routines "userdie" and "usernice", aliases for "die" and "nice". "Userdie" takes an argument that

is placed in "diesmesg" so that "killoff" can provide "inclear" (Section 5.3) with it.

The procedure "killoff" first removes the target process from the ready queue, if it is there. The status is set to "halted". If "killoff" was invoked to halt the process, it then returns. (This feature is not currently used.) If "killoff" is to terminate the process, it then invokes "intclear" to remove any interrupt handlers associated with the target (Section 6.1), "freerel" to return the target's stack (and the data area for Elmer processes), "inclear" in "inmaint.u" to clear the target's link table (Section 5.3), "kmclean" in "kmesg.u" to remove any messages awaiting the target (Section 5.4), and then it reduces the reference count that records how many processes are active in the target's core image. If the process is an Elmer process, its data area is returned to free storage. The routine "inclear" can itself indirectly cause the termination of other processes, so "killoff" is indirectly recursive. A running process cannot be halted or terminated, so "killoff" changes the state of a running process to "to be killed" or "to be halted", as appropriate. Each service call invokes the procedure "maydie", which checks to see if the current process is in one of these states. If so, "schedcall" is invoked to return control to the scheduler, which can then invoke "killoff" to finish the action. In case the process was slated for termination, "killoff" sets "diesmesg" to "killed".

Both "awaken" and the scheduler loop use locks (Section 3.6)

to prevent corruption of the ready queue.

The scheduler is itself started by the routine "startscheduler". First, the routine "scheduler" is initiated as if it were a normal process. This action provides the scheduler with a reasonable stack. Then "strtsur" is invoked to transfer control to the scheduler. Its first action is to invoke "kill-off" on itself, removing all vestiges of itself from the process table. The "killoff" routine knows not to remove the scheduler's stack.

Files

schedule.h, schedule.u, resume.s

Data Structures

STKLEN

Length of a user stack, in words.

int NUMPROCS

Size of the process table.

int idtable[NUMPROCS]

Table of process identifiers for each process number.

int curusr

Process number of currently running process.

int curerror

If not zero, the current service call has raised an exception.

int curtype

Indicates if current process is C or Elmer, so that the service call dispatcher can handle service calls correctly.

int nextid

Non-reusable id for next process to be initiated, stored at absolute location 157784.

char *diemsg

Either 0 or points to a message provided by the current user in a userdie call or by the kernel in terminating the user.

int kernno

Process number of kernel job; set by lsl.c

struct pnode {

int ppfptr; /* stack frame pointer: stored r5 of user program */

int ppstatus; /* One of the following:

PPNONEX #1

PPSLEEPING #2

PPREADY #4

PPRUNNING #10

PPHALTED #20

PPTOHALT #40
PPTOKILL #100

int pnext; /* for linking into queues: in [0..NUMPROCS-1] */
int *ppstklim; /* stacktop (lowest address allowed). Also address of stack for allocation */
int ppcodeno; /* index into codetab, -1 if resident. */
int pptype; /* 0 if C program, 1 for Elmer */
int ppdata; /* location of data area of Elmer program */
char *pperror; /* location of user error handler, or 0 */
int *ppcatch; /* address of message catcher */
int ppcdata; /* catcher data area */
int ppcurmess; /* catcher urmess area */
int ppallocatch; /* bitstring of all channels caught. Used to determine if a channel is caught, NOT ppcatch! */
int ppcpend; /* true if a catch is pending */
int pptime[TIMESLOTS]; /* number of 10,000ths of secs used in this interval */
};

struct pnode proctab[NUMPROCS]

per-process data; indexed by proctab

int schedfp

stack-frame pointer for the scheduler

int schedstklim

stack limit for the scheduler

int schedno

process number of the scheduler until it wipes it out

int rqlfirst, rqlast

head and tail of the ready queue

int curusr

Process number of currently running process.

int NUMMODULES

Length of core image table.

struct codetentry {

int cdrefcount; /* reference count (-1 if not in use) */

char *cdmemory; /* start address of this segment */

int cdlength; /* number of words in the text part of the segment */

int cdchksum; /* checksum of the text part of the segment

in cddata; /* 0 for C, otherwise length of data area for Elmer */

int cdowner; /* process id of owner */

} codetab[NUMMODULES]; The core image table.

Procedures

maydie()
If current process is in state "to kill" or "tohalt", calls "schedcall".

startscheduler()
Called during initialization. Starts up the scheduler as a process, then transfers to it.

killoff(userno,how)
The second argument is either PPROKILL or PPTOHALT. Halts or terminates the target process. If the process is running, sets the status to PPROKILL or PPTOHALT. If the process gets terminated, removes its stack, link table, data area (Elmer only), waiting messages, and interrupt handlers.

int newseg(start,owner,data,length) int *start
Finds a free slot in "codetab", initializes it so that its "cdmemory" field holds "start", and returns the slot index. Computes the checksum of the text part of the code segment and stores it in "codetab". Data is the number of words in the data area for Elmer, 0 for C. Length is the size of the text region, in words.

int getcodeseg(userno)
Returns the core image stored in the proctab.

int getfreeno()
Finds an available user number.

fallthrough()
Procedure called if a process returns. Calls "schedcall(DIE)".

int initiate(codeseg,arg,owner,fd)
Creates a new process in codeseg, creating the appropriate stack space to this process. Codeseg names the entry in codetab that governs this code segment. If the codeseg is for Elmer, then fd must be a file descriptor for the source file, so the data segment can be read. Makes a new process entry in proctab. Its entry point is the start of the memory in the given core image, and the stack is initialized to hold the given argument. The process is left in state "ready".

scheduler()
First removes itself from the process table, retaining its own stack. Then enters a loop in which a ready process is scheduled and run through "strtusr". Upon return (through "schedcall"), process may be rescheduled, terminated, or neither, for the case CONTINUE, DIE, and SLEEP.

awaken(procno)
Cause the given process to be placed on the runnable queue if it was sleeping.

userdie(msg) char *msg
This service call is invoked by the kernel call "die(msg)" and effects "schedcall(DIE)". It also puts msg in "diemsg" for communication with Inclear.

usarnice()
This service call is invoked by the kernel call "nice" and effects "schedcall(CONTINUE)".

char *uerthandle(addr) char *addr
This service call establishes the "pperror" field in "proctab" for this user, and it returns the previous value.

char *gerthandler()
Returns the "pperror" field in "proctab".

schedinit()
Initializes local tables.

int getno(procno)
Finds the process number for the given process id.

int getid(procno)
Finds the process id for the given process number.

strtusr(fptr,result,chkflag) int *fptr, *result, chkflag
in "resume.s". Starts the process whose frame pointer is given. The scheduler stack pointer is saved in fptr. When this routine returns (the user process has called "schedcall(arg)", "arg" is returned through "result". If "chkflag" is TRUE, then "maycatch" in "kmes.s" is called to treat any asynchronous message receipts that are pending (Section 5.5).

schedcall(kind)
In "resume.s". Switches back to scheduler stack frame, places "kind" as a result, and returns from what looks like the call to "strtusr" made earlier.

checkmem(cdptr) struct codentry *cdptr
Returns the checksum of the given text space. In a userdie call or by the kernel in terminating the user.

4.6 Core images

The module "lifetime.u" provides service calls that allow one process to control another. A process can cause a program to be loaded into memory by invoking "userload" (an alias for "load", which takes a file name and a file descriptor. The file descriptor is a link to an open file. (Links are described in Section 5.3, and the file handler is described elsewhere [6,7].)

The program "userload" calls routine "uload" in module "uloader.u" to bring in the core image of the file. This module is written as a user program, using standard service calls to accomplish the necessary communication with the file manager to read the file. If the file descriptor is -1, then "uloader"

tries to read the file directly over the fast line to Unix. In this case, the file name is used as a console prompt; otherwise, the file handler is employed. The routine "uload" uses a privileged service call to acquire free space for the new load image. By inspecting the first bytes of the file, it can determine whether the file is Elmer or C. In the former case, only the code part is read in. In the latter case, the code part is read, relocation is performed, initialized data are loaded, and room is reserved and cleared for uninitialized data. The loader returns the starting address of the loaded program. The service routine "userload" finishes by establishing a code segment for the new image by calling "newcseg" in "schedule.u" and returning the index of the code segment that "newcseg" provides.

The caller of "userload" can then use this image number to start a new process in the loaded image. Since loading and startup are independent, the calling process (usually the resource manager) may start several processes in the same image and may save the images of terminated processes in which new processes may be started later. Starting a process is accomplished by the service call "userstart" (an alias for "startup"), which takes a code segment, an argument, a link to the parent, and a disposition code for that link. This routine calls "initiate" to start up a new process in that code segment (if the owner is right). If the core image to be started is an Elmer program, then "userstart" requires an extra argument: a link to the file system for the data area of the program. "Userstart"

calls "uload" in "uloader" to read in Elmer data areas.

The new process is started with a fresh link table that contains one link, the parent link supplied to "userstart". (See the discussion of links in Section 5.3.) This link is either duplicated for the child or given away outright, depending on the disposition. This link has the restriction "NODESTROY", so the child cannot destroy it except by terminating. (The parent can thus be furnished an unforgeable notification of the child's termination.) A lifeline is then created for the parent. It appears in many ways like a link in the parent's link table, except that it is not possible to send a message along it, and it has the restriction bit LIFELINE. The destination of the lifeline is the new child.

A kludge to allow remote loading of programs checks the file name in the "userload" call if the file descriptor is good. If the file name is a small integer between 0 and NUMMACH, the number of machines, then the call is taken as a request for remote loading. In this case, two extra arguments are used, one to be the parent link of the new process, and one to be the argument to that process. A special message is sent to the kernel job on the destination to request loading. Since the process id of the kernel job on a different machine is not universally known, a special case is made. A process id having the machine number in the upper byte and a zero in the lower byte can always be used to refer to the kernel job on that machine. Protocols for the kernel-to-kernel message are in "kernkern.h". The kernel job on the remote machine not only loads the program, it also starts it

with the supplied parent link and argument. The remote kernel job returns the lifeline to the started process by sending its response along a link that it builds to the requesting process with channel 15. The routine "userload" waits for this response, then returns the lifeline to the calling process.

A process holding a lifeline may use it to control the owner of the lifeline (that is, the process to which it points). The service routine "userkill", alias for "kill", takes a lifeline as an argument. This routine sends a KILL notification to the target process. (Eventually, "userhalt" and "userresume" may be added.)

In order to remove a core image, a process may invoke the service routine "userremove" (an alias for "remove") in "schedule.u". This routine reclaims the memory through "freerel" and frees the slot in the segment table. The caller must be the owner of that segment, and the reference count must be 0.

Files
lifeline.u, lmain.h, uloader.u, kernkern.h, kerjob.u,
schedule.u.

Procedures

```
int userload(prog,fd) char *prog
    Load in a new program from the given file. Either use the
    file descriptor, or the file name. If prog-1, load remotely
    and return lifeline. Else return the image number of the
    new core image.
int userstart(codeseg,arg,plink,disp)
    Start a process in the given image with the given argument.
    Initialize it to have the given parent link. Return a life-
    line to the new process.
int userkill(lifeline)
    Send a KILL note to the process pointed to by the lifeline.
    userremove(codeseg)
    If the caller owns the code segment, and the reference count
    is 0, reclaim the storage for the segment. This routine is
    in "schedule.u".
    uload(fname,startloc,length,fd)
```

```
char *fname; char **startloc;" int *length;
    Loads the named file (which must be in Unix a.out format).
    If startloc = 0, it is a fresh file, and startloc will be
    set to a new area to report where the file was put. Length
    will return the size in words of the text portion of the
    file. If startloc is anything else, then it is assumed that
    only the data area is to be read, starting where ever start-
    loc requires. Fd is a link to a file system process. This
    link is closed, even on failure.
```

4.7 Timing

Each process has an array of TIMESLOTS (currently 5) entries that record recent CPU usage. This array stored in the process table (Section 4.1) as "pptime". Each entry refers to an interval of one second, and the number stored there is the number of 10,000ths of a second that the given process ran during that interval. Information is only retained for the previous TIMESLOTS seconds. These arrays are indexed by the variable "timeslot". Just before the scheduler starts a process, it records the current time in "starttime" to the nearest 10,000th of a second. When the process returns to the scheduler, that start time is subtracted from the current time, and the result is added to the appropriate entry in the timing array.

The module "timing.u" contains routines for gathering timing statistics for processes. The routine "nexttime" is called from the clock (Section 3.8) every second. This routine increments "timeslot" and clears the array entry associated with the new second for each non-running process. For the currently running process, "nexttime" adds the increment that has been used in the current second into the old timeslot before advancing the index. It also puts the current time in "starttime" so that if the pro-

cess should return to the scheduler during the current interval, the scheduler will add the correct increment to the timing array.

The routine "userdsp", an alias for "display", may be invoked by a process to get timing statistics about another process. The target process is named by a link (Section 5.3). If it is on a different machine, "userdsp" returns an error. If not, "userdsp" totals the time used in the last TIMESLOTS-1 complete seconds and returns the percentage of the time (from 0 to 100) that the target process used the CPU during that interval.

Files

timing.h, timing.u

Data Structures

TIMESLOTS

The number of seconds of information stored for each process, currently 5.

int timeslot

An index into the pptime array in each pptime.

long int starttime

when the current job started

Procedures

nexttime()

A second has elapsed. Increments timeslot mod NUMSECS and clears the pptime word in each process. Special action for the current process: Add remnant of current second into its timing before incrementation, and change starttime.

int userdsp(linknumber)

Called by a user process. Returns the fraction of the last TIMESLOTS slots that was used by the process at the destination of the linknumber. This answer will be an integer between 0 and 100. Returns -2 if the destination does not exist or is on a different machine. Causes usererror -1 if the given link is bad.

5. MESSAGES

All communication among processes is carried out through the medium of messages. Three major modules in Arachne deal with message handling. The module "line.u" (and the small module "route.u") deal with messages sent to foreign sites or arriving from foreign sites. The module "message.u" contains the service calls "receive" and "send". The central message-handling module is "kmesg.u".

5.1 Central message handling

The module "kmesg.u" maintains a pool of unused message buffers each of which has slots for the message text, the message length (from 0 to MSLEN), routing information (destination, channel, code), a note field that indicates the purpose of the message (for example, DATA or DESTROYED), an enclosed link (Section 5.3), and a pointer field used for linking unused buffers into a queue. All access to the pool of message buffers is protected by locks (Section 3.6).

Message buffers can be acquired by invoking the routine "getkmesg" and released by "rikmesg". The former routine takes a priority argument. If the priority is 1, then any available buffer is returned. If the priority is 2, then a buffer is only given if there are at least 1/4 of the original buffers free. If the priority is 3, then a buffer is only given if there are at least 1/2 of the original buffers free. These distinctions are

used to implement flow control. The callers to "getmesg" are in the other two message modules.

Each process has a queue of messages waiting for it, arranged in the order they arrive. This queue is protected by locks (Section 3.6). Messages are placed on the queue by the routine "sendit", which is invoked by both the other message modules. This routine places the message on the appropriate queue if its destination is local to this machine. (The destination is a process id, which includes the machine id. A process id that has lower byte 0 is interpreted to refer to the kernjob, whatever its proper process id may be.) If the destination is no longer alive, the message is discarded. If the destination is on a foreign machine, then "sendit" finds the appropriate line on which to send the message by calling "getline" in the module "route.u", then calling "sendblock" in the module "line.u" to ship it off (Section 5.2). The routine "sendit" recognizes one special case: If the "note" field of the message to a local process is the code "KILL" or "HALT", then instead of delivering the message, sendit invokes "killoff" in module "schedule" with argument TOKILL or TONALT. (Section 4.5. HALT is not currently used.) In the case of KILL aimed at a non-running process, the "diemesg" mechanism that provides owners of its links with meaningful destruction notices does not work, since "diemesg" is global and "sendit" may run at interrupt level, "diemesg" is not used.

The dual to "sendit" is "waitmess", which is invoked by module "message.u" to get a message from the queue for a process.

The caller specifies a set of channels. The routine "waitmess" will return the first message on the appropriate queue whose channel is one of those specified. The caller also specifies a timeout period. If the timeout is 0 and no appropriate message is waiting, then "waitmess" returns failure. If the timeout is negative and no message is waiting, the routine "waitmess" calls "schedcall(SLEEP)" to allow other processes to carry on. Every time "sendit" deposits a message in a process queue, it invokes "awaken" to inform that process. If the destination was not asleep, then "awaken" has no effect. If the message was not the one expected, then the process returns to sleep. Eventually, the process that is sleeping for a message will be awakened and will be able to continue. ("Schedcall" and "awaken" are described in Section 4.5.) If the delay is positive and no appropriate message has yet arrived, then "waitmess" uses the clock routine "setalarm" (Section 3.8) to awaken the sleeping process after that amount of time. If an appropriate message arrives first, the alarm is turned off. When the alarm rings, "waitmess" is awakened, discovers that no message has arrived, and returns failure to the caller. The routine "waitmess" uses "time" in the module "clock.u" to distinguish between an alarm and the awakening that accompanies the arrival of a message.

The length field on the message dictates how many characters are copied from it into the process' buffer.

In some cases of user error, the module "message.u" must give back a message it has taken. In this case, "giveback" is

invoked to put it at the head of the queue.

When a process is killed, "killoff" in "schedule.u" calls "kclear" in "kmess.u" to remove any message that might be queued up for the process that is dying. The buffers occupied by the messages are reclaimed.

Files kmess.h, kmess.u

Data Structures

```
struct kmesg {
    int knext; /* links together free list */
    int klength; /* number of bytes of message in kmbody */
    int kmdst; /* destination process id */
    int kcode, knote, kmchan;
    struct link klinkenc; /* enclosed link */
    char kmbody[MSLEN]; /* MSLEN defined in lmain.h */
}
```

One kernel message buffer.

int kmesgsize
Size of a kmesg in words. Set at Arachne initialization.

struct kmesg *kmesgbuf
Initialized to kmesgbuf[NUMKMES].

struct kmesg *kmbufavail

Head of available list.

int numkmes, warn1, warn2

Number of buffers left, 1/2 the original number of buffers, 1/4 the original number of buffers.

NUMKMES

Original number of kernel message buffers available

NUMWTMES

Number of waiting messages that can be held before users take them.

struct kmesg kawaited[NUMPROCS]
Headers for each process into a list of kmesgs waiting for receipt, each linked through the knext field.

Procedures

struct kmesg *getkmesg(priority)

Returns a pointer to a free kmesg buffer. The meaning of priority is described above. Returns 0 on failure (if no buffer is available at this priority).

rlkmesg(kmesg) struct kmesg *kmesg

Returns the kmesg buffer to the free buffer pool.

kmainit()

Initializes all tables for kmess.u.

sendit(kmesg) struct kmesg *kmesg

Sends a message to the destination indicated in the message. If the message looks foreign, try to route it through an ap-

propriate neighbor.

```
int waitmess(who, chans, delay, kmgot) struct kmesg *kmgot
Wait for a message for user number "who" on any of the chans indicated in the mask "chans". Delay is a maximum delay (in seconds), after which a return of -1 is given if no message was received. A delay of -1 indicates no time limit. kmgot is a result parameter, set to point to the received kmesg buffer.
```

```
giveback(kmess) struct kmesg *kmesg
```

Place the indicated message back at the head of incoming messages for the current user (curusr).

```
kaclear(who)
```

Remove any incoming messages on the queue for process number "who".

5.2 Inter-machine Messages

The modules "line.u" and "route.u" provide low-level communication with other machines. "Route.u" associates physical lines with processor id's by the routine "getline". The actual handling of physical lines is in "line.u".

"Line.u" keeps tables for each physical line indicating the current state of the communication. Input and output lines are completely independent. For input lines, the information stored includes the address of the device registers, the state of the transmission (idle, waiting for a free buffer, receiving the message size, and receiving the message body), a pointer to the message buffer holding the incoming message, and a count of how many bytes are left to transmit. Output lines carry much the same information in addition to a queue of message buffers awaiting transmission. No acknowledgment or checksums are used, since experience has shown that the lines are very reliable. All access to the outgoing message queues is protected by locks (Section 3.6). All the queues are locked simultaneously for simpli-

city.

During initialization, this module determines which physical lines to neighbors exist by attempting to read the status register for each line. If the register does not exist, the processor traps, and this trap is caught by "setflag" in "lowestirp.s". This routine sets a flag that is examined by line initialization.

The routine "sendblock" places a given message buffer on the appropriate output queue and calls "outfrob". "Outfrob" signals "linehandle" by clearing and setting the output interrupt enable bit on the appropriate line. This toggling is performed at high priority. If the line is currently not engaged in output, this action will cause an output interrupt. If it is, then an output interrupt will occur in any case as soon as output is finished.

Input interrupts on DRV-11 lines are dispatched to "inhandle", and output interrupts to "outhandle". (See Section 3.2 on interrupt handling.) These routines take an argument that indicates which line caused the interrupt. Interrupts are serviced based on the current state of communication on the particular line that caused an interrupt.

An output interrupt on an idle line causes "linehandle" to examine the queue of outgoing messages on that line. If it is not empty, then the first is picked and readied for sending. An input interrupt on an idle line causes "linehandle" to prepare to receive a foreign message. It calls "getkmesg" in "kmesg.u" with priority 3 to find a message buffer in which to place the incoming message. If this request fails, then "linehandle" does not accept the first word of the incoming message. In this case, the

machine trying to send the message will not be able to use the line until the recipient machine is able to find a buffer and reads the header word from the line. The procedure "frob" is used by "rlkmesg" in module "kmesg.u" to cause input interrupts on those lines waiting for a message buffer whenever a buffer becomes available. Once the header word has been accepted, the sending side of the line sends the entire message buffer. During the bulk of the transmission, both "inhandle" and "outhandle" try to stay locked in the interrupt routine in order to use the physical line at peak efficiency. If the receiver fails to receive at a reasonable rate, then the sender dismisses the interrupt, but will come back when the next transmitted word has been read.

Files

line.u, route.u, and line.h

Data Structures

```
int linetab[NUMHACH] (in route.u)
    Linetab[n] is the number of the line to machine n.
struct drvblock {
    int drcsr; /* common status register */
    int droutbuf; /* output data buffer */
    int drinbuf; /* input data buffer */
    int drfiller; /* not used */
}
```

The structure of a block of registers pertaining to one physical DRV-11 (parallel-word) interface to another LSI-11.

Number of physical links to neighboring LSI-11's. Set during initialization.

```
struct inblock {
    int *indir; /* the drv block address */
    int instate; /* place in protocol for this DRV-11 line */
    struct kmesg *incurmess; /* where output incoming stuff */
    int *inptr; /* points within incurmess */
    int inwcnt; /* number of words left to transfer */
    intinfo [NUMHRS]
    struct outblock {
        int *outdr; /* the drv block address */
        int outstate; /* place in protocol for this DRV-11 line */
        struct kmesg *outcurmess; /* kmesg currently in transit */
        int *outptr; /* points within outcurmess */
    }
}
```

```

int outdent; /* number of words left to transfer */
struct kmsg *loouthead, *loouttail; /* head, tail of
output queue, linked through knext field.
} outinfo [NUMBERS] Data indicating the state of one DRV-11
line.
struct intervect {
int *outnpc, outnps, *innpc, innps; /* new pc and ps for
output and input interrupts */
} interrupt vector block for one DRV-11 line.

```

Procedures

```

routeinit()
Initializes linetab.
int getline(machine)
Returns the line corresponding to machine "machno". Aborts
on an invalid machine number.
char *irpvect(machine)
Returns the address of the interrupt vector for machine
"mach".
char *drv(machine)
Returns the address of the device register block for machine
"mach".
irpinit()
Initializes the states (ioblocks) of all lines.
sendblock(mach, block) struct kmsg *block
Attempts to send the message pointed to by "block" to the
neighbor whose machine id is "mach". If there is no room on
the output queue, sets an alarm to try again later, but re-
turns for now.
outfrob(dr)
Causes an output interrupt on the DRV-11 line whose address
is "dr".
infrrob()
Causes an input interrupt on those DRV-11 lines in "waiting"
state.
inhandle(machine)
Services an input interrupt that occurred on the line from
machine "machine".
outhandle(machine)
Services an output interrupt that occurred on the line from
machine "machine".

```

5.3 Links

Each process has a table of links that are used to send mes-
sages. The table of links, "lntab", and all manipulations of
links are handled in the module "lntab.u". A destination field

of s indicates that an entry is not in use.

A new link can be created by the service routine "lnmake",
an alias for the service call "link". This routine returns a
small number that the process can use to refer to this link;
processes never have direct access to the link table. The new
link is initialized with a destination pointing to the calling
process, code, channel and restriction according to information
provided by the calling process. The code and channel are pro-
vided to the process when it receives any message that comes
along that link to help it identify the purpose of the message.
The channel can also be used for selective receipt of messages.
The restrictions are a set of permissions and actions that will
govern the use of this link. The owner (the process that created
the link) can either allow or prohibit the holder (a process that
is given the link in order to send messages to the owner) giving
the link away or duplicating it. The owner can request notifica-
tion when the link is duplicated, given away, or destroyed. None
of these restrictions applies as long as the link is PRISTINE,
that is, was not received in a message or duplicated from another
link. Notifications are special messages along the channel and
code of the link with the unforgeable note field indicating what
notification type it is. The restriction bits also indicate
whether the link is a use-once resource (a REPLY link) or a per-
manent resource (a REQUEST link). New processes are given a link
(always numbered 0) that their parent presented to the kernel.
This link is restricted by NODESTROY, so the child may not des-
troy the link and fool the parent into believing that the child

has terminated. Finally, the MAYERROR restriction bit allows error messages (Section 5.4) to be sent on the link.

Lifelines (Section 4.6) are like links, except that LIFELINE is set as a restriction. The owner cannot create a link with LIFELINE on or PRISTINE off. It is not possible to send a message across a lifeline.

After a link has been created, it can be given to another process as an enclosure in a message. Making a link can overflow the space in intab for the process. Overflow can also result from receiving a link in a message. The routine "lnget", called to find a free space in this table, will expand the table if necessary. The initial allocation is added on each time an overflow occurs, up to a fixed limit. The table space is reserved from free space (Section 3.5).

The routine "lnclear" is called from "killoff" in "schedule.u". It sets the destination field of all links in the current process' link table to 0, clearing them. If any link has the TELLEST (tell upon destruction) bit set, then "tell(DESTROYED,die,msg)" is invoked. If any link has the LIFELINE bit set, then "tell(KILL)" is invoked to kill the destination process. If any link has MAYERROR but not TELLEST set, then "tell(ERROR,"holder died")" is invoked to warn the destination process.

The service routine "lndestr", an alias for "destroy", is used to destroy links. It clears the destination field after sending any necessary notifications by using "tell". It refuses to destroy a link with NODESTROY set. ("lnclear" is the only way

to destroy such a link.)

The service routine "lnok", an alias for "linkok", returns 0 if the given link number is currently valid; otherwise a negative error code is returned.

Files lnmaint.u and lnmaint.h

Data Structures

```
struct link {
    int lncode;
    int lndest; /* destination. 0 if link not in use */
    int lnrestr; /* restriction bits */
    int lnchan; /* channel */
}

struct lntentry {
    int lntsize;
    struct link *lntaddr;
} lntab[NUMPROCS];

/* A table of all links for all processes.

int NUMLINKS
    Number of initial links per process.

int MAXNUMLINKS
    Maximum number of links per process.
```

Procedures

```
int lnmake(code, chan, restr)
    Make a new link, with destination pointing to the calling process, with the given channel, code, and restrictions.

lnclear(userno,msg) char *msg;
    Remove all links for this user. If any have the TELLEST restriction, send a DESTROYED notification to the destination, with "msg" as the body. If any has the KILLABLE restriction, kill the destination with a KILL notification. If any has MAYERROR but not TELLEST, warn the destination with an ERROR notification.

telldup(dup,pin) struct link *pin;
    If the given link has TELLDUP or TELLGIVE restrictions, and dup is DUP or NODUP, respectively, then a notification is sent along this link.

lncopy(to,from)
    Copies the link, setting PRISTINE off.

lndup(lulink)
    Duplicate the given link, and return the index of the new one. If the link has the TELLDUP restriction, send the DUPPED notification to the owner.

lndestr(ulink)
    Destroy the given link, but return an error code if not possible. If the link has the TELLEST restriction, send the DESTROYED notification to the owner.
```

```

int lnfree(userno,ln) struct link **ln
Return the index and address of a free link in the given
user's link table, if possible.
int lndecode(alink,ln) struct link **ln
Check to see if "alink" is a valid link number for the
current user. If so, return a pointer to the link table en-
try via the result parameter "ln". Otherwise, return -1.
lnmake(ln) struct link **ln
Makes an entry in the caller's link table that contains this
link. This procedure is used by kernjob to forge links.
lninit()
Initialize all tables for lnmain.u.
int lnget(who,where) struct link **where;
Returns the link number of an unused link for process "who".
Sets "where" to point to the link itself. Increases the
size of the process' link table if necessary, but not beyond
MAXLINKS. Returns -1 for failure.
int lnstart(who)
Sets up an initial link table for process "who" with NUM-
LINKS links, all with null destinations. Returns -1 on
failure.

```

5.4 User messages

The user interface to message passing is contained in the module "message.u". The service routines in this module exten- sively check consistency of arguments given by calling processes. For example, pointers to memory are checked against "outrange" in module "isl.c". If any errors are found, the service routine first undoes any work it may already have performed (which may involve returning a message buffer via "giveback" in "kmess.u") and returns to the caller by using the macro "USERERROR" (Section 3.3).

The service routine "sendumes", an alias for the user ser- vice call "send", checks that all the actions desired by the caller are in consonance with the permissions of the link across which the message is to be sent. If all is well, "sendumes" ac- quires a message buffer from "getkmesg" in "kmess.u". The prior-

ity argument is 1 if the message is traveling on a reply link and 2 otherwise. (Replies are more likely to be actively awaited by their destinations than are requests, which may build up.) If "getkmesg" refuses to allocate a message buffer, then the calling process waits for one in a "schedcall(CONTINUE)" loop inside "trygetkmesg" in "message.u". Once a message buffer is avail- able, the contents of the caller's message (if any) are copied into the message buffer, along with the destination field ex- tracted from the link along which it is being sent and any link to be enclosed. Only as much of the message as specified in the length field is copied into the kernel message buffer. If the disposition has ERROR set and the carrier link has MAYERROR, then the note field of the message buffer is set to ERROR; otherwise, it is set to DATA. The message buffer is then given to "sendit" in "kmess.u" to direct the message toward its recipient. The link on which the message was sent and the enclosed link are then removed from the sender's link table if this action is called for by the restrictions on the carrier link and the disposition argu- ments, respectively. If the restrictions on these links dictate, notifications are sent to their destinations.

All notifications are sent by the routine "tell" in "message.u", which immediately calls "telling", which takes as arguments the destination, channel, code, and notification type. If "telling" cannot get a message buffer at priority 2, it sets an alarm on the clock to try again in one second. By the time the alarm rings, the original link along which the notification is to be sent may have disappeared, but the necessary information

from it is stored as the arguments to "telling".

The dual service routine "recumess", an alias for "receive", is used to receive messages. The caller indicates which collection of channels to use and where to put the contents of the message when it arrives. The message body is placed in a location specified by the caller; other information is placed in a "urmess" structure provided by the caller. This structure contains fields for the length, the note, and the channel and code describing the link used. The calling process can request a timeout after which the call should fail if no message has arrived. This argument is passed directly on to "waitmess" in module "kmess.u". After the contents of the message have been copied into the process data area, the message buffer is reclaimed with "rlkmesg" in "kmess.u". If the note on the message was ERROR, then an exception is raised (Section 6.2), but otherwise the receipt is successful.

Files

message.u and message.h

Data Structures

```

struct urmess
/* what a user receives */
int urlength, urcode, urnote, urchan;
/* length of body, user-defined code, system-given note
(DUPPED, DESTROYED, GIVEN, ERROR, DATA) */
int urlinenc; /* enclosed link index into ur linktab */
}; /* end of struct urmess */

```

Procedures

tell(who, what, mesg) struct link *who; char *mesg
 Send a message over link "who" with note "what", one of DUPPED, DESTROYED, ERROR, GIVEN, and KILL. Uses routine "telling". In case of DESTROYED and ERROR, "mesg" is passed to "telling" as well.
 telling(dest, chan, code, what, mesg) char *mesg;
 If can't send a note immediately, set an alarm to try again in a second.
 mscopy(to, from) char *to, *from
 Copy body of message. If from = 0, fill "to" with nulls.
 int sendmess(u link, data, length, disp) char *data
 Send message "data" with length "length" over the link numbered "u link" in the current process' link table. If u link is the index of a valid link in the current process' link table, create a link in the destination process' link table equivalent to it. Remove u link from the sending process' link table. If disp does not have DUP set, the message has note DATA unless disp has ERROR set. Return 0 for success, a negative error code for failure.
 int recumess(chans, data, urmess, delay) struct urmess *urmess
 Receive a user message on the channel combination specified by "chans". Place it into the user-provided buffer "data". Extra information (length of message, channel, code, note) is placed in "urmess". If delay >= 0 and no message is received in "delay" seconds, fail. Return 0 for success, a negative failure code for fail. The timeout failure code is -3. If the note of the received message is ERROR, terminate the caller.
 struct kmesg *trygetkmesg(priority)
 Call getkmesg(priority) until a message buffer is obtained. Perform "schedcall(CONTINUE)" while waiting.
 telldup(disp, pin) struct link *pin
 If disp has the DUP bit set, tell owner of link pin DUPPED; otherwise, tell owner GIVEN. In latter case, remove pin from table by setting its destination to 0.

5.5 Asynchronous Message Receipt

A process may arrange for messages arriving on certain channels to be asynchronously received as soon as they arrive. This arrangement is established by a call to "usercatch", which is an alias for "catcher". The arguments indicate the channels that are to be treated specially, the address of the procedure that is to be called asynchronously, and addresses of data structures

into which the message is to be placed when that procedure is called. If the address of the catcher procedure is 0, then instead of adding the given channels to the active set, those channels are removed from it.

"Usercatch" establishes asynchronous receipt by initializing several fields in the "ppnode" structure in the process table (Section 4.1). The address of the catcher is placed in the "ppcatch" field, the message contents address in "ppcdata", and the urmess address in "ppurmess". The field "ppallicatch" is set to hold a bitstring naming all the channels that are active for asynchronous receipt. Only one catch procedure may be specified at any time; a new specification overrides the old one.

Whenever "sendit" in "kmess.u" delivers a message, it checks to see if it should be caught asynchronously instead of delivered. If so, the process target process is either currently running or not. If it is running, then "docatch" in "message.u" is called to dispatch the message to the catcher routine. "Docatch" sets "curlevel" to 1 to imply that an interrupt-level routine is running. This value prevents the catcher procedure supplied by the process from using most of the service calls. The "awaken" service call is the only exception; it makes a special check to insure that it is not used during a catch. The catcher procedure itself is called without arguments. If it returns TRUE, then the channel remains active; otherwise, the channel on which the message arrived is deactivated for asynchronous receipt.

If the target process is not currently running, then the

"ppcpend" field in the process table is set to indicate that the next time the process is run, "docatch" should be called. The last argument to "strtuser" in the scheduler is the value of the "ppcpend" field. If it is TRUE, then "strtuser" will call "maycatch" in "message.u" before continuing the process from its previous place. This procedure calls "docatch" for each message on the input message queue that should be received asynchronously.

Procedures

```
int usercatch(chans,data,urmess,caproc)
    char *data; struct urmess *urmess; int *caproc;
    If "caproc" is 0, removes the given channels from the set
    that are active for asynchronous receipt. If "caproc" is
    not 0, then the channels named will be activated for catching
    messages. In addition, "data", "urmess", and "caproc"
    become the new standard way to treat messages that are
    caught.
```

```
docatch(kmess) struct kmess *kmess;
```

Called if this message is one that the current user wished to catch. Calls the user's catch routine, and either disables or re-enables the channel for catching depending on how that routine returns. During the catch routine, set "curlevel" to 1 to prevent most service calls from being used. The awaken service call is a special case, treated there.

```
maycatch()
```

Sees if any message on the input message queue for the current process fits any of the catchers. If so, invokes the appropriate catcher for each one.

6. INTERRUPTS AND EXCEPTIONS

Processes can inform the kernel that they wish to handle their own interrupts. Likewise, exceptions that are raised may be handled by the offending process.

6.1 Interrupts

The module "interrupt.u" contains the routines that accomplish interrupt dispatching. A process can call the service routine "userhandler", an alias for "handler", naming an interrupt vector, the address of the routine that should service interrupts arriving through that vector, and a channel number. The routine "userhandler" sets the interrupt vector to jump at high priority to a routine in "lowestirp.s", one of "uint0", "uint1", up to the number of allowed interrupt handlers. Each of these routines places an argument (one of 0, 1, ...) on the stack and calls the routine "uinterrupt" in "interrupt.u". This routine therefore receives an argument telling which of the possible interrupts has happened. It then calls the appropriate user-supplied interrupt routine. When that routine finishes, "uinterrupt" returns, and the interrupt is dismissed from "uint0" or whichever one it came through.

While the process is handling an interrupt, it may invoke the service routine "userawaken", an alias for "awaken", which takes no arguments. This routine, in "interrupt.u", causes a message to be directed to the process that created the handler. The correct process is found through the variable "curhandler", which is set by "uinterrupt" when the interrupt is dispatched. The message is on the channel specified by the process when it invoked "userhandler", with note INTERRUPT. The code is 0, and the body of the message is empty.

No other service calls may be called by a process interrupt

handler. Such calls will be aborted by "sys" in "crti.s".

Files

interrupt.u and lowestirp.s

Data Structures

int NUMHDLs

Number of handlers that may exist.

int numhds

current number of handlers in existence

struct hdlnode {

int (*hentry)(); /* entry point of user handler */

int hchannel; /* channel for awaken calls */

int *hdvector; /* vector for the interrupt */

int hdest; /* destination for awakens. 0 means this

handler not in use */

};

struct hdlnode hdlset[NUMHDLs], *curhandler

Set of handlers, current handler in force.

int *lowuint[]

In lowestirp.s: pointers to individual uint routines

Procedures

Intclear(procid)

Clears all interrupt vectors owned by this process. This operation is part of killing a process.

userhandler(vector,entry,channel) int *vector, *entry

Service call to set up a handler at given vector, with given

entry point. The channel is used in an associated

"userawaken" call.

uinterrupt(hindex)

Called from "uint?" in "lowestirp.s" when an interrupt oc-

curs. Dispatches to appropriate handler.

int userawaken()

Service call to be used only from an interrupt handler.

Causes a notification to be sent to the process that created

the handler.

uintinit()

Initialization of the interrupt table.

6.2 Exceptions

Exceptions are raised when a process makes a service call that cannot be completed because the arguments are bad. Service call routines use the macro USRERROR when the caller has made a mistake. This macro calls "usererror" in "error.c", which prints

a message and sets "cureerror" to 1.

Also, messages may be sent that contain error notifications. If the "dup" argument to "sendmes" has ERROR set, and the link along which the message is to be sent has MAYERROR in the restriction bits, then an error message is sent. Receipt of an error message (by "recmes" in "message.u", Section 5.4, but not by asynchronous receipt, Section 5.5) will also set "cureerror" to 1.

After any service routine completes, control is returned to the caller through the wormhole at "sys" in "crt1.s". This code checks "cureerror" to see if the service call completed normally. If not, then the exception can have one of two effects: If the caller has taken no special precautions against exceptions, then the wormhole calls "userdie("exception not caught")", terminating the process.

If the caller has arranged for exceptions to be caught, then the exception handler provided by the process earlier is called. The arguments to this procedure are the returned value of the service call, the service code, and then the arguments to the service call during whose execution the exception was raised. The exception handler may take any action it wishes; when it returns, control passes to the point to which the service call would have returned. Any value that the exception handler returns becomes the visible result of the service call.

A user process may request that all exceptions be caught in this fashion by invoking "uerrhandler" in "schedule.u", an alias for "errhandler". This procedure takes one argument: the address of the error handler procedure. If it is 0, then "uerrhandler"

removes any existing error handler. The address of the current error handler is stored in the "pperror" field of the process table entry for each process (Section 4.1). The wormhole discovers this address by calling "gerthandler" in "schedule.u".

Files

schedule.u, error.c, crt1.s

Procedures

char uerrhandler(addr) char *addr;
Establishes a new error handler. Returns the old handler address.
char gerthandler()
Called from "crt1.s". Returns the current error handler.

7. ACKNOWLEDGEMENTS

The authors are pleased to acknowledge the assistance of Professor Sun Zhong Xiu, visiting scholar from Nanking University (Peoples Republic of China), as well as students Jonathan Dreyer, Jack Fishburn, Will Leland, Paul Pierce, and Ronald Fischler. Their hard work has helped Arachne to reach its current level of development.

8. REFERENCES

- [1] M. H. Solomon and R. A. Finkel, "ROSCOE -- a multi-processor operating system," Technical Report #321, University of Wisconsin-Madison Computer Sciences (September 1978).
- [2] M. H. Solomon and R. A. Finkel, "ROSCOE: a multi-processor operating system," Proceedings of the Second Rocky Mountain Symposium on Microcomputers, pp. 291-310 (August 1978).

gust 1978).

- [3] M. Solomon and R. A. Finkel, "The Roscoe Distributed Operating System," Proceedings of the Seventh Symposium on Operating Systems Principles, pp. 108-114 (10-12 December, 1979).
- [4] R. L. Tischler, R. A. Finkel, and M. H. Solomon, "Roscoe User Guide, Version 1.0," Technical Report #336, University of Wisconsin--Madison Computer Sciences (September 1978).
- [5] R. A. Finkel, M. H. Solomon, and R. Tischler, "Roscoe User Guide, Version 1.1," Technical Report #1938, University of Wisconsin--Madison Mathematics Research Center (March 1979).
- [6] R. A. Finkel, M. H. Solomon, and R. Tischler, "Arachne User Guide, Version 1.2," Technical Report #379, University of Wisconsin--Madison Computer Sciences (February 1980).
- [7] R. A. Finkel, M. H. Solomon, and R. L. Tischler, "Roscoe Utility Processes," Technical Report #338, University of Wisconsin--Madison Computer Sciences (February 1979).
- [8] R. A. Finkel, M. Solomon, and R. and Tischler, "The Roscoe Resource Manager," Proceedings of Compcon Spring 1979, pp. 88-91 (February, 1979).
- [9] B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall (1978).
- [10] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," Communications of the ACM 17, 7, pp. 365-375 (July 1974).
- [11] R. A. Finkel and M. H. Solomon, "The Roscoe Kernel, Version 1.0," Technical Report #337, University of Wisconsin--Madison Computer Sciences (September 1978).
- [12] DEC (Digital Equipment Corporation), Microcomputer Handbook (second edition), 1976.
- [13] D. E. Knuth, The Art of Computer Programming Volume 1--Fundamental Algorithms (second edition), Addison Wesley (1973).

APPENDIX G

Roscoe Utility Processes

Ron Tischler
Raphael Finkel
Marvin Solomon

Technical Report 338
Computer Sciences Department
University of Wisconsin-Madison
February 1979

ROSCOE UTILITY PROCESSES*

February 1979

Ron Tischler
Raphael Finkel
Marvin Solomon

Technical Report 338

Abstract

Roscoe is a multi-computer operating system running on a network of LSI-11 computers at the University of Wisconsin. Roscoe consists of a kernel program resident on each computer and several utility processes. This document describes the implementation of the Roscoe utility processes at the level of detail necessary for a programmer who intends to add a module or modify the existing code. Companion reports describe the purposes and concepts underlying the Roscoe project, present the implementation details of the kernel, and display Roscoe from the point of view of the user program.

*This research was supported in part by the United States Army under contract #DAAG29-75-C-0024.

TABLE OF CONTENTS

1.	THE RESOURCE MANAGER.....	2
1.1	General.....	2
1.2	Protocols between resource managers.....	2
1.3	Resource manager initialization.....	4
1.4	Wall clock synchronization.....	7
1.5	Process initiation.....	7
1.6	Process termination.....	10
1.7	Terminal links.....	11
1.8	FOREGROUND processes.....	11
2.	THE TERMINAL DRIVER.....	16
2.1	General.....	16
2.2	Overview of input.....	17
2.3	Overview of output.....	18
2.4	Requesting and changing console modes.....	18
2.5	Output buffer manipulation.....	19
2.6	Input buffers.....	20
2.7	Pause control.....	23
2.8	Control-C actions.....	24
2.9	Pausing and continuing.....	25
3.	THE COMMAND INTERPRETER.....	27
3.1	General.....	27
3.2	Initialization.....	28
3.3	Command line parsing.....	28
3.4	Command execution.....	29
4.	THE FILE MANAGER.....	32
4.1	General.....	32
4.2	Execution of requests.....	33
5.	THE DEMON.....	35
5.1	General.....	35
5.2	DALIAS command.....	37
5.3	DCLOSE command.....	37
5.4	DCREAT command.....	37
5.5	DOPEN command.....	37

5.6	DREAD command.....	37
5.7	DREADLINE command.....	38
5.8	DSEEK command.....	38
5.9	DSTAT command.....	38
5.10	DTIME command.....	38
5.11	DUNLINK command.....	39
5.12	DWRITE command.....	39
6.	LIBRARY ROUTINES.....	39
6.1	File manager routines.....	39
6.2	Resource manager request routines.....	41
6.3	Roscoe service calls.....	42
6.4	Miscellaneous.....	42

ROSCOE UTILITY PROCESSES

This paper documents the source code for the following Roscoe utilities:

- resource manager
- terminal driver
- command interpreter
- file manager
- the "demon" (a PDP-11/40 process with which the file manager communicates)
- user-callable library routines
- copyfile (a program used implicitly by the command interpreter)

The reader is assumed to be familiar with the Roscoe User Guide [Tischler, Solomon, and Finkel 78], which describes the purposes and use of these utilities. The present paper consists of a detailed explanation of the programs and data structures for those who intend to help maintain these utilities. The Roscoe kernel code is similarly documented [Finkel and Solomon 78].

The documentation given here is accurate as of January 20, 1979. However, recent developments will soon cause some modifications to the processes discussed here. In particular, a new utility process called a "pipe" has been introduced to attach the output of one process to the input of a second one. The command interpreter and the resource manager will cooperate to establish piped processes.

Unless otherwise stated, all files mentioned are in the directory `"/usr/network/roscoe/user"`.

1. THE RESOURCE MANAGER

1.1 General

The code lies in "resource.u". Programs that communicate with the resource manager should include "resource.h" unless all such communication is handled by library routines.

The resource manager may use the service calls "load", "remove", "startup", and "kill". These calls are meant to be privileged, although that restriction is not yet enforced. The "startup" call gives the new process a link to its resource manager. This link should be of a special kind, although the resource manager currently refers to it as a "REQUEST" link. Either a REQUEST or REPLY link may be enclosed over this link. Currently, the kernel does not enforce the restrictions on enclosed links. Furthermore, the new process may not destroy this parent link except by dying. This restriction is enforced.

1.2 Protocols between resource managers

When resource managers talk to each other, they send requests whose "rmreq" fields hold special values. These values are defined by macros that begin with the letters "RR". We will follow the convention of calling the originator of such a message the "first" resource manager and the recipient the "second". Together, they are called "colleagues". When client processes talk to resource managers, they send requests whose "rmreq" fields are

defined by macros that begin with the letters "RM". Following sections describe how these codes are employed to carry out the various resource manager functions.

The routine "sendrm" is used to send messages between resource managers. The array "rmtab", of size 5, keeps track of which other resource managers exist. Entries in "rmtab" are link numbers; -1 indicates that there is no corresponding resource manager. Links used between resource managers use channel 2, and the code is always the machine number of the holder.

Resource managers may make RMFSREQ or RMTTREQ requests of each other, in which case the request is treated the same as any other user's request, except that for RMTTREQ, the local terminal link is assumed to be the one desired. In addition, there are five other requests peculiar to resource managers, as listed below. Whenever these requests are made, the first resource manager does not wait for a reply; any reply that eventually comes will be self-explanatory.

RRSTART: This request continues an RMSTART request that the first resource manager could not complete. Everything in the original request is passed along; no response is needed. The resource managers pass the request around in order of increasing machine id. The originator recognizes it should it return. This circular method is an ad-hoc approach that will be changed in the future to a more reasonable polling order.

RRKILL: This request continues an RMKILL request if the process targeted for the kill is not local to the first resource manager. The request is forwarded to the resource manager on the

proper machine; no response is needed.

RRLINK: This message asks the second resource manager for a link owned by that second resource manager. The first resource manager intends to give this link to a third resource manager.

RRINFORM: This message accompanies an enclosed link owned or held by the first resource manager. (See Section 1.3.)

RRPASS: Used to "pass the ball" when a FOREGROUND process "with the ball" for a certain terminal has died, and the process that should next "get the ball" is on another machine. (See Section 1.8).

1.3 Resource manager initialization

When a resource manager is loaded by the kernel job of the Roscoe kernel, it receives the machine number as the argument to "main". In particular, if the bit "NOTPAPA" is off, this resource manager knows that it is the first one. We will call such a resource manager "original". A resource manager that the kernel job starts in an attempt to recover from failure at some node or as a subsidiary resource manager has the bit NOTPAPA set.

When the original resource manager starts, initialization is done by "initrm0". A file manager and terminal driver are loaded and started as DETACHED processes; the file manager is loaded manually, and does not occupy a spot in "imagetab". Input and output terminal links are opened, a "configuration" is read from the terminal by "readline", and the input link is closed. The "configuration" is a character string that the resource manager scans to determine what other resource managers to load and with

what arguments. For example, the configuration

1T24FT

indicates that resource managers should be loaded on machines 1, 2, and 4; machine 1 will have its own terminal, machine 4 will have its own terminal and file system, and machine 2 will have neither.

The argument given to a remote resource manager has the machine number as its lowest three bits and contains flags RMTTFLAG and RMFSFLAG to indicate respectively whether a terminal driver (and attendant command interpreter) and file manager should be loaded locally. Also, the bit "NOTPAPA" is set to indicate that the child resource manager is not the first one started. The high order byte of the argument gives the machine number of the parent (papa).

When a resource manager other than the original one starts, it uses the initialization routine "initrms". An entry is made in "rmtab" for the first resource manager (the owner of this resource manager's link 0), and an RRINFORM message is sent to that resource manager with an enclosed link having channel 2. The code for this link is the number of the first resource manager. The "rmarg" field of this initial message is -1. (The discussion of RRINFORM messages continues below.) If the RMFSFLAG is on, a local file manager is loaded by the routine "loadfs", which asks the first resource manager for a file system link, uses it to perform the load, and then destroys this unneeded link. If the RMTTFLAG is on, a local terminal driver and command interpreter are loaded. If these flags are off, the ap-

propriate links are obtained from the first resource manager by the same RMTTREQ and RMFSREQ protocols followed by any other process.

The routines "loadtt" and "loadci" are used to load local copies of the terminal driver and command interpreter, respectively. No matter how a terminal is obtained, a terminal output link is automatically opened. The variable "owntt" tells which terminal (0-4) the resource manager is using. The command interpreter is vaccinated against control-C's by setting its "lifeno" field in "proctab" to -1.

The routine "rrinform" handles RRINFORM requests. If the "rmarg" field is -1, the receiver knows that a new resource manager just came to life. The number of this new resource manager can be found in inmess.urcode. The receiver then acts as the "papa" and sends out RRLINK messages to begin the process of hooking together all the other resource managers. Otherwise, the high order byte of the "rmarg" field tells the number of the resource manager that owns the link, and the low order byte tells for which resource manager the link is intended. If this intended holder is not the present resource manager, the RRINFORM request is forwarded to the correct one. Whenever a resource manager receives a link which it will continue to hold, it updates its "rmtab" accordingly.

The routine "rrlink" handles RRLINK requests which, as mentioned above, are only sent by the original resource manager to other resource managers. A link is created on channel 2; the code is specified by "rmarg", which indicates the resource

manager that will eventually hold the link. The link is sent to the first resource manager in an RRINFORM message; it will then forward it to the intended holder, as described above.

1.4 Wall clock synchronization

When the original resource manager starts, a special request is sent to the demon on the PDP-11/40 for the Unix date. The variable "timewarp" is used to convert between Roscoe time and Unix time (the former begins Jan 1 1973 CST; the latter Jan 1 1970 GMT). Other resource managers initialize their dates to zero, but this value is soon corrected.

Whenever "sendrm" is used (to send an RRSTART, RRKILL, RRINFORM, RRLINK, or RRPASS message to a colleague), the current date is placed in the "update" field of the message. Whenever such a request is received, the local date is set to the value in the "update" field if it is later. This algorithm keeps the wall clocks in the various Roscoe kernels from losing time relative to each other.

1.5 Process initiation

The resource manager knows which client sent each RMSTART request because the code of the link containing the request is also the index for that process in the resource manager's process table. The resource manager can also determine the client's associated terminal from this table. If the request cannot be processed locally (either the "load" or "startup" service call fails due to lack of room), then an RRSTART request is sent to the resource

manager with the next higher machine number (modulo 5), as determined from "rmtab". The RRSTART request has all the information of the RMSTART request (including the same enclosed link, if any), plus the client's process identifier and terminal number, which would not otherwise be known to the second resource manager. The second resource manager tries likewise to initiate the child process, and if it also fails, sends the request on further. The identifier of the child includes its machine number as its lowest three bits; if the RRSTART returns to the client's resource manager, it is recognized as a failed request. It may be sent around once more (if the original method involved the GENTLY mode) but, in any case, the buck stops somewhere, either with success or failure. A reply (if required) is sent to the client from the resource manager where the algorithm stops.

The routines "rmstart" and "rrstart" are invoked for RMSTART and RRSTART requests, respectively. Each of these routines computes the client's process identifier and terminal number in its own way and then calls "rawstart". Another argument to "rawstart" tells whether the load should have GENTLY or ROUGHLY mode. An RRSTART message received at the client's machine is recognized by "rrstart"; if the mode was GENTLY, "rawstart" is now called with ROUGHLY mode; if the mode was ROUGHLY, a negative reply is sent to the client. If the load doesn't succeed locally and there are no other machines, "rawstart" similarly calls itself with mode ROUGHLY (if the mode was GENTLY), or sends the user a negative reply (if the mode was ROUGHLY).

The routine "getimage" loads a program. A "stat" checks that

the file is a publicly executable load-format file and computes its date of last modification. If no acceptable copy already resides locally, a new one is loaded. If "imagetab" is full, an unused image is removed (in ROUGHLY mode). The procedure "make-room" is used to remove unused images. Unused images are also removed (in ROUGHLY mode) until there is room for the new image to be loaded. Images are removed in ascending order of their index in "imagetab". When a new image is loaded, "imagetab" is updated accordingly. Future developments should prevent loading if a colleague has a useable copy, and removal of images should perhaps use some other algorithm.

The routine "newproc" starts a process. The new process is given a link to the resource manager on channel 1; this link has type REQUEST and TELLDEST, and its code is the new child's index in "proctab". If the start succeeds, the corresponding "count" field in "imagetab" is incremented, and an entry is made in "proctab". If the start fails because there was no room for the process's stack, then "makeroom" is called, as in the case of "getimage" described above. The lowest three bits of the child's process identifier tell the machine number; the variable "uniquecode" generates unique process identifiers. If the child is to be "DETACHED", its lifeline is destroyed.

1.6 Process termination

The "rmarg" field of an RMKILL request tells the process identifier of the victim. The resource manager figures out which colleague hosts the victim by looking at the lowest three bits, and then either completes the request itself or forwards an RRKILL request to the appropriate colleague. The RRKILL request contains the process identifier of the client, which otherwise wouldn't be known to the colleague and which is needed to check that the client has permission to kill the victim. The routine "rawkill" checks this permission and then performs the kill; the victim's lifeline isn't destroyed (yet). There's nothing to prevent the parent of a FOREGROUND process from performing a kill if it correctly guesses the child's process identifier.

When a client terminates, naturally or otherwise, the resource manager receives a DESTROYED message on its link and calls "procdie". The client's entry in "proctab" is deleted by setting the "proctype" field to UNUSED, except for FOREGROUND processes (see further discussion below). The corresponding "count" field in "imagetab" is decremented. The process's parent link and/or lifeline are destroyed if the resource manager still holds them.

The termination of a colleague is similarly detected. The routine "rmdie" updates "rmtab" accordingly.

1.7 Terminal links

When a request for a terminal link is received over channel 1, the corresponding "ttyeno" field in "proctab" is compared to "ownntt" to see if the local terminal link is desired. If not, the routine "gettt" is used to ask the appropriate colleague for a copy of its terminal link. RMTTREQ requests received over channel 2 (i.e., from a colleague) are always given a copy of the local terminal link.

1.8 FOREGROUND processes

One linked list of FOREGROUND processes is associated with each terminal; at most one terminal is owned by each resource manager. The process that "has the ball" (will be killed by the next Control-C) is at the head of this list, and it points to the next process to "get the ball". Segments of this list reside physically on each machine; each list logically threads its way among several machines.

Two fields in a process table entry are relevant to this discussion. The field "parentno" is the process identifier of the process's parent; the last three bits of this number tell the parent's machine number. The field "parentp" is the index in "proctab" of the next local item in the FOREGROUND list. When a process's successor is on the same machine, "parentp" points to it, and the last three bits of "parentno" are the machine id; when a process's successor is on another machine, the last three bits of "parentno" tell which machine to go to next, and

"parentp" tells which local process comes next when the chain returns to this machine. Each terminal chain in each resource manager has a special header node containing two fields: "foretop" gives the index of the first item in the process table (i.e., it corresponds to a "parentp"), and "theball" is a Boolean that tells whether this machine (i.e., the process indicated by "foretop") "has the ball". A null pointer value for "parentp" or "foretop" is indicated by -1.

When a resource manager receives an RMSTART request with FOREGROUND mode, "rmstart" checks that the client "has the ball" and turns off its "theball" flag. If the load doesn't succeed locally, "rawstart" sends an RRSTART message as usual. Wherever the load succeeds, the routine "newproc" will turn on the "theball" flag, and insert the new process at the head of the appropriate list. The routine "tellt" is used by "rawstart" to send a TOKILL message to a terminal driver (local or not), so the terminal driver will know which process now "has the ball". If the start fails, the resource manager that initiated it notices the request returning (perhaps for the second time); its "theball" flag is turned back on by "rrstart", so the process that previously "had the ball" still does.

When a FOREGROUND process dies, "procdie" marks the corresponding "proctype" entry in "proctab" as DEFUNCT, rather than UNUSED. The "parentno" and "parentp" fields are still relevant, so the item is still linked up. These DEFUNCT items are cleaned off as FOREGROUND processes die in the "proper" sequence. Specifically, when the process "with the ball" dies,

"procdie" turns off the "theball" flag and calls "rrpass" to clean off DEFUNCT processes at the head of the FOREGROUND list so long as they point to other processes on the same machine. If a non-DEFUNCT process is reached in this manner, it then "has the ball"; the "theball" flag is turned on, and "tellt" is used to send an appropriate TOKILL message to a terminal driver. On the other hand, if the list points to another machine, an RRPASS message is sent to the appropriate colleague. When a resource manager receives an RRPASS message, it also uses "rrpass" to clean off DEFUNCT processes and/or "pass the ball".

Files

resource.u, resource.h

Data Structures

```
struct rmmesg { /* messages to resource managers */
    int rmreq; /* type of request */
    int rmarg; /* various miscellaneous arguments */
    int rmmode; /* the mode for STARTs or KILLs */
    long update; /* time field used between R.M.'s */
    int parno; /* parent's proc. id., used by R.M.'s */
    int ttno; /* a terminal number, used by R.M.'s */
} rmmesg; /* contents of outmess, used implicitly */

struct { /* image table entry */
    char fname[RMFNAMESZ]; /* file name */
    long loadtime; /* time it was loaded */
    int count; /* number of active processes */
    int procmode; /* SHARE, REUSE, or VIRGIN */
    int imageno; /* image, used for "start" or "remove" */
} imagetab[NBRIMAGES]; /* image table */

struct proctab { /* process table entry */
    int proctype; /* FOREGROUND, BACKGROUND, DETACHED,
UNUSED, or DEFUNCT */
    int parentp; /* an index in this table */
    int parentno; /* process identifier of the parent */
    int plink; /* link supplied by parent during start */
    int location; /* index into the image table */
    int lifeno; /* lifeline, used for "kill" */
    int procid; /* its process identifier */
    int ttyeno; /* its terminal number */
} proctab[NBRPROCS]; /* known process table */
```

Procedures

main(arg)

Initializes the resource manager for machine given in "arg",

executes a loop that receives and dispatches requests. The argument also contains the bit NOTPAPA.

respond(n)
Sends a one-word response to the client. If it is a negative error indicator, destroys the link that client submitted.

giveaway(elink,how)
Returns the "elink" to the current client. "How" is either "DUP" or "NODUP" to govern the disposition of that link.

int sendrm(n,elink,how)
Sends a message to another resource manager. "n" is the index of the resource manager to send it to, "elink", "how" describe the link to enclose; "elink" = NOLINK means to really not send a link. Returns 0 on success, -1 on failure; sometimes the caller cares.

cryout(message) char *message;
General-purpose error indication routine.

gettt(n)
Asks the resource manager on machine n for its terminal link. Returns either the link or -1 for error.

telltt(tt,elink)
Tells terminal on link "tt" that its new killlink is "elink".

loadfs()
Gets a file manager link from the original resource manager and uses it to load a file manager on this machine.

loadtt()
Loads a terminal driver on this machine and gets an output link to it.

loadci()
Loads a command interpreter on this machine; assumes there is already a terminal driver.

initrm0()
Initialization specific to the original resource manager. Finds the local time from Unix, loads the first file manager via a manual load, loads a terminal driver, gets an input line to ask for the configuration, then decodes the configuration and loads the other machines.

initrms(arg)
Initialization specific to non-original resource managers. Informs the original resource manager; either loads a local file manager and terminal or uses links to the original resource manager's copies.

rmstart()
Handles client request to start a new process. If FOREGROUND, insures the client currently has the ball. Calls "rawstart".

rrstart()
Handles request from another resource manager to do a start. If the request has come full circle, either gives up or tries roughly. Calls "rawstart".

rawstart(parent,tttype,how)
Tries to start a process on this machine. "Parent" gives the procid of the client who initiated it all, "tttype" gives

its teletype number, "how" is GENTLY or ROUGHLY, but matters only inside "getimage". Calls "newproc".

rmkill()
Handles client request to kill a process. Either directs the request to the appropriate resource manager or calls "rawkill".

rawkill(parent)
Checks if the parent has the right to submit this kill request; if so, submits a "kill" service call.

rrlink()
The papa resource manager has requested a link for a third party. The third party's number is in "contents->rmarg"; the papa's number is "inmess.urcode". Prepares a link and sends it.

rrinform()
Handles a new link given by a colleague. Establishes knowledge of that colleague in the proper tables. During recovery actions, the new information may disagree with the old.

rrpass()
This resource manager has just been given the ball. Cleans off the defunct part of the foreground stack, and if it becomes empty, sends the ball elsewhere.

rmddie()
Just found out that a colleague has died. Clears out entries in "rmtab".

int getimage(name,mode,how) char *name;
Loads a new core image, and returns an index into imagetab. If "how" = ROUGHLY, will also try to make room; otherwise just hopes there is room, or a usable copy exists. The mode is SHARE, REUSE, or VIRGIN. Returns -1 if the load fails. Checks that the image is executable, and will not use an existing image with obsolete date. Discovers if the image is an Elmer program.

int newproc(imagno,arg,parent,type,ttype)
Makes a new process by using the service call "startup". Returns an index in the updated proctab or -1 for error. "Imagno" is the index in imagetab for the process's image, "arg" is the argument to give the new process, "parent" is the parent's procid, "type" is BACKGROUND, FOREGROUND, or DETACHED, "ttype" tells which terminal to use. If the process is in Elmer, opens its object file in order to give the link to "startup".

procdie()
Cleans up after the termination of a client. If it was in the foreground and had the ball, the ball is passed, possibly to a colleague.

int getindex(i)
Returns the index in proctab for the process whose procid is i.

int makeroom()
Makes room in the image table if possible, and returns the index of an available slot. If necessary, core images of terminated processes are removed. Returns -1 on failure.

```

drwrite(word)
    Busy-waits until the DR-11 line to Unix is ready, then
    writes one word.
int drread()
    Busy-waits until the DR-11 line from Unix is ready, then re-
    turns one word read.

```

2. THE TERMINAL DRIVER

2.1 General

The code lies in "ttdriver.u". Processes using the terminal driver should include "ttdriver.h" and "filesys.h". (The latter contains macros used by both the terminal driver and file manager.) It isn't necessary to include these if all communication is done by library routines.

The terminal driver gives its parent (usually a resource manager) a REQUEST link on channel 1. All requests to open the terminal for input or output come over this link or copies thereof; also, the resource manager sends "TOKILL" messages over this link. (See Section 2.8.) An "input link" is used for "read" and "readline" requests from the client; an "output link" is used for "write" requests. Thus, the terminal driver accepts data on its output links and supplies data on its input link. At most one input link and NUMCODES (currently 5) output links may be open. (Channel 2 is used for output, channel 3 for input.) These links are of type GIVEALL but not DUPALL. Destruction of such a link is interpreted as a "close" command. The holder of the input link may also use it to request or change terminal modes. (See Section 2.4.)

Input and output are interrupt-driven; the routines "ttin-driv" and "ttoutdriv" are invoked at interrupt level when the corresponding devices become ready. Initially, interrupts are enabled and "handler" calls set up the interrupt vectors. The interrupt-level routines send "awaken" messages to the terminal driver on channel 10. Interrupts are disabled at crucial times by turning off the appropriate interrupt-enable bits in the device registers. (Since interrupt-level routines run at high priority, this interrupt disabling is not strictly necessary.) These interrupt-driven routines share data with the rest of the terminal driver.

2.2 Overview of input

The Boolean variable "inuse" tells whether an input link is open. The routine "readmsg" executes a "read" or "readline" request by calling "getchar" for one character at a time. A Boolean value is returned by "getchar" to tell if the character terminates the current line; if so, the character returned is either a newline, control-D, control-W, or null. The first three of these are appropriately interpreted (depending on whether the command is "read" or "readline"); the meaning of a null is explained in Section 2.8. At most MSLEN characters may be read at a time; thus the reply will fit in one message.

2.3 Overview of output

The integer arrays "codes" and "bytesleft", each of size NUM-CODES, keep track of output links. A zero entry in "codes" indicates an unused link; otherwise links are given unique codes. When a link is opened, the corresponding entry in "bytesleft" is set to zero; when a write message is received, the indicated length of the write is placed in "bytesleft". Subsequent messages over this link are interpreted as data to be written until the write is completed. The routine "writemsg" is used to write each portion; characters are written with the routine "sayfull" (Section 2.5), except for carriage returns and newlines, which are written directly by calling "sayit".

2.4 Requesting and changing console modes

The variable "modes" stores the current modes. When modes are requested or changed, the routine "showstate" prints out the "current" or "new" modes, respectively. The modes that can be turned on and off are ECHO, HARD, UPPER, and TABS. A HARD terminal cannot backspace its cursor legibly; and UPPER terminal cannot enter lower case directly, and a TABS terminal has hardware tabs.

2.5 Output buffer manipulation

The terminal driver uses a circular output buffer, "ttoutbuf", of size TTOUTBUFSIZE (150). Two variables are used as indices into "ttoutbuf": "nxtoutch", which points to the next available place to put a character into the buffer, and "outbufp", which points to the next character to take out. When these indices are equal, the buffer is empty.

The routine "ttoutdriv" is called at interrupt level to write a character. This routine returns without any action if "paused" is true (Section 2.7). If "ttoutbuf" is nonempty, the next character is written to the terminal, with a delay specified in absolute location 157702 (for debugging and connection to a slow Unix port). After it is displayed, each newline is replaced in the buffer by a carriage return rather than being removed; by this device, carriage returns are effectively appended to newlines. Input interrupts are disabled while messing with the buffer (in case ECHO mode is on).

The routine "sayit" puts one character into "ttoutbuf". If doing so would fill the buffer, "sayit" waits a second and tries again. The variables "outpos" and "tablptr" are significant for echoing input; "sayit" sets them both to zero after a carriage return or newline. In other cases "outpos" is incremented, except that after a backspace it is decremented. While the buffer is being changed, input and output interrupts are disabled (we might be in ECHO mode).

The routine "sayfull" converts a character into a readable

(or audible) form and calls "sayit". If UPPER mode is on, a "!" is placed before appropriate characters. Except for control-G (bell), control characters are converted to the notation "^A", etc. A rubout is converted to "^#". The array "tabplace" is used to store the cursor position just before each tab, to allow backspacing over tabs; "tablptr" is an index in "tabplace". At most TABNUM (10) tabs are stored. If TABS mode is off, a tab is converted into several blanks, until "outpos" becomes a multiple of 8 ("sayit" increments "outpos"). If TABS mode is on, the tab character is sent directly to "sayit", and "outpos" is adjusted accordingly.

The routine "sayback" is used when the console is in ECHO but not HARD mode. It converts a given character into several backspaces, to undo the effect of "sayfull", and calls "sayit". Two backspaces are required for control characters, (escaped) rubout, and the UPPER mode escape sequences, except that none are needed for bells. Other character take one backspace, except for tabs, which require a sequence of backspaces until "outpos" has been decremented (by "sayit") to the appropriate value found in "tabplace". Also, "tablptr" is decremented.

2.6 Input buffers

The terminal driver uses a circular input buffer, "ttinbuf", of size TTINBUFSIZE (100), and a circular buffer, "lineptr", of size TTLINES (20). The entries in "lineptr" are indices in "ttinbuf" that tell where lines begin; thus TTLINES is the maximum number of lookahead lines. There are two other variables

used as "ttinbuf" indices: "inbufp", which tells the next available spot to put a character into "ttinbuf", and "nxtchar", which tells the next character to take from the buffer. When "nxtchar" is one buffer location ahead of "inbufp", the buffer is full. To permit intra-line editing, lines can only be removed from the buffer when they have been "terminated". There are two variables used as "lineptr" indices: "lastline", which tells the current line being put into the buffer, and "nxtline", which tells the line being removed. The buffer is empty when "lastline" and "nxtline" are equal.

The routine "ttindriv" is called at interrupt level to read a character. If "paused" is true, then only control-Q and control-S will have any effect; all other characters are ignored (Section 2.7). The Boolean variable "escaping" is true when the next character is to have no special meaning; it becomes true when the escape character is read and becomes false after the following character. A character with no special meaning is placed in the buffer by calling "putinbuf"; if the buffer is full, the character is discarded and a bell is written by calling "sayit". If there is room and ECHO mode is on, the character is written by calling "sayfull"; for example, an escaped newline will echo as "^J", which allows backspacing over it later. If UPPER mode is on, appropriate translation takes place.

Various characters cause intra-line editing. A control-C is echoed as the sequence

AD-A123 586

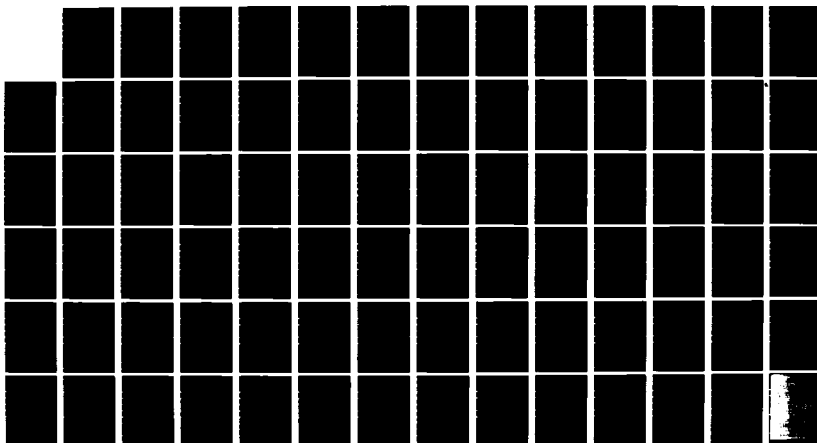
OPERATING SYSTEMS FOR RING-BASED MULTIPROCESSORS(U)
WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES
R A FINKEL ET AL. 1982 N00014-81-C-2151

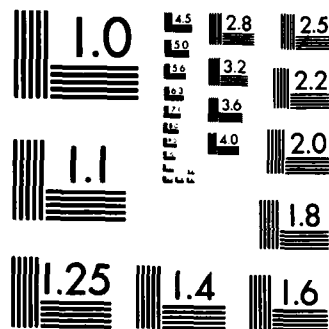
4/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

`^C<bell><newline>`

and isn't placed in the input buffer; see Section 2.8. An ERASE character (rubout) causes the last character of the present line to be removed from the buffer, using the routine "tkoutbuf". If the line wasn't empty and ECHO mode is on, the character removed from the buffer is echoed, using "sayfull" in HARD mode and "say-back" otherwise. In HARD mode, backslashes ('\') are placed around a sequence of erased characters; when erasing begins, a backslash is echoed and the Boolean variable "erasing" becomes true; when erasing ends, a backslash is echoed and "erasing" becomes false. A KILL character (control-X) removes the entire present line from the buffer. In ECHO mode, "??" is printed. In HARD mode, a newline is also printed. If we are in ECHO but not HARD mode, the KILL is treated as a sequence of ERASEs, until the current line is empty; thus the screen cursor returns to the point where the line began.

A line is "terminated" by an (unescaped) control-D, control-W, carriage return, or newline. In the first two cases, the character is put in the buffer but not echoed. In the last two cases, a newline is put in the buffer and echoed. The routine "termline" updates "lineptr"; if this buffer is full, the line termination is ignored. Whenever a line is terminated, the variable "linecount" is reset to 0. This variable keeps track of how many lines have been output to the terminal since the last time the user entered a line. An "awaken" call is made in case "getchar" (described next) was waiting for the buffer to become nonempty. If this awaken is received in the terminal driver's

main loop, it is properly ignored.

The routine "getchar" takes a character out of "ttinbuf" and also returns a Boolean value to tell if a line has just been completed. (Thus, for example, escaped and unescaped newlines are distinguished.) When a line is completed, "lineptr" is appropriately updated. If the buffer was empty, "getchar" waits for a message on channel 10 to indicate that the interrupt level routine "ttindriv" has put a line into the buffer. (This message might also indicate a control-C; see Section 2.8.) Input interrupts are disabled when the buffer is in an awkward state.

2.7 Pause control

Pause control uses the commands control-S and control-Q. Initially, "scroll" is false. If a control-S is typed in this state, "scroll" is set to true and "pause" is also set to true. The effect is that output pauses until released, and it will continue to periodically pause every SCROLLLEN (18) lines. When the terminal is paused, a control-S will cause it to be released for the next 18 lines, but a control-Q will release it and turn off scroll mode, so it will not stop again. Control-Q can also be used to turn off scroll mode even if the terminal is not currently paused.

2.8 Control-C actions

The variable "killlink" contains the lifeline along which a kill should be performed upon receipt of a control-C. Initially, "killlink" is set to -1 to indicate the absence of such a lifeline. The resource manager encloses such a lifeline in a "TO-KILL" message to the terminal driver, received on channel 1. When a lifeline is received, the Boolean variable "ctrlC" is set to false.

The interrupt level routine "ttindriv" notices when a control-C is typed. If "ctrlC" was false and "killlink" was non-negative, "ctrlC" is set to true and a message is sent to the terminal driver on channel 10 with an "awaken" call. While "ctrlC" is true, all messages received on channels 2 and 3 are ignored and any links enclosed in such messages are destroyed.

The message on channel 10 is received either in the terminal driver's main loop or in the routine "getchar", which was waiting for a non-empty input buffer. In either case, the routine "chkctrlC" performs the kill if "ctrlC" is true, flushes all outstanding messages on channel 10, and reinitializes "killlink" and the input and output buffers. Any lame-duck messages on channels 2 and 3 will be ignored because "ctrlC" is still true. A Boolean value is returned by "chkctrlC" so that "getchar" knows that the message was a control-C indicator rather than a non-empty buffer indicator.

After a kill is performed, control must return to the main loop. If the message had been received inside "getchar", the

value returned is the null character as a line terminator. This special case is recognized by "readmsg" which then returns to the main loop without completing its read request; the link enclosed with that request is destroyed.

2.9 Pausing and continuing

When an unescaped control-S is read, the variable "paused" is set to true. When an unescaped control-Q is read, that variable is reset to false and the output interrupt enabling is toggled to restart the output-interrupt driven routine "ttoutdriv". That routine returns without any action if "paused" is true.

Files

ttdriver.u, ttdriver.h, filesys.h

Data Structures

```
char ttinbuf[TTINBUFSIZE]
    Circular input buffer filled by ttindriv, emptied by
    readmsg.
int lineptr[TTLINES]
    Circular buffer of ttinbuf indices that point to beginnings
    of lines.
char escaping
    Boolean; set by ESCAPE, reset by next character.
char erasing
    Boolean; true during a sequence of ERASEs; only used in hard
    copy mode.
char modes
    Bits used: ECHO, TABS, HARD, UPPER.
int tabplace[TABNUM], tabplptr
    Remembers where tabs were.
char ttoutbuf[TTOUTBUFSIZE]
    Circular output buffer. Filled by sayit, emptied by ttout-
    driv.
int codes[NUMCODES]
    Currently active output lines.
int bytesleft[NUMCODES]
    Used to keep track of pieces of different write messages.
int killlink
    Tells the ttdriver whom to kill on ^C.
```

Procedures

main(dev)

Initializes tables, provides parent with a request link,

prepares to use terminal whose device register is at "dev". Executes a loop that receives and dispatches client requests.

`readmsg(len,how)`
 Reads "len" characters, using routine "getchar". At most MSLEN characters are read. Reading terminates if a control-D is read. In the case that "how" is READLINE, any line terminator (control-W or <cr> or <lf>) terminates reading.

`char getchar(ch) char *ch;`
 Gets a character from the input buffer and returns it in "ch". The returned value is Boolean: TRUE means the character returned ends a line.

`ttindriv()`
 Called by "ttinint" when a character is ready; runs at interrupt level. Reading a control-C causes an "awaken" service call, ERASE or KILL cause intra-line editing. Line termination is caused by control-D, control-W, <cr> (converted into <lf>) and <lf>. Termination causes an "awaken" service call. The input buffer is updated, and the input is properly echoed.

`char putinbuf(ch) char ch;`
 Puts the given character into the input buffer. It returns TRUE only if there was room in the buffer.

`char tkoutbuf(ch) char *ch;`
 Removes last character of current line from buffer. Returns TRUE only if something was there. The character is returned in "ch".

`termline()`
 Called at interrupt level to cause an "awaken" and to reset buffer pointers.

`resetbuf()`
 Removes the current line by resetting an input buffer pointer.

`writemsg()`
 Decodes a write message from a client. If it is the header of several packets with data, variables are initialized to receive the data. If data have arrived, they are placed in the output buffer by "sayit" and "sayfull".

`char chkctrlC()`
 If a control-C has been received, all awaken messages are flushed, a service call "kill" is performed along the killlink, and the routine returns "TRUE".

`closeinput()`
 Reduces the count of input lines in use.

`closeoutput()`
 Resets the appropriate output line information.

`openline(how)`
 Handles a client request for a new input or output line, as described by "how". Appropriate variables are initialized.

`reply(retcode,size) char retcode;`
 Reports "retcode" to the current client. The "size" parameter tells how much of the standard message buffer has been filled with other useful information that the client must

also receive. The reply code is put in the first byte.

`showstate(when) char *when;`
 Prints the current modes on the terminal with an introductory message determined by "when".

`sayit(ch) char ch;`
 Puts one character in the output buffer and adjusts position variable "outpos" accordingly. This routine is used both for input and output echoing.

`sayfull(ch) char ch;`
 Uses "sayit" to provide a readable form for any character according to the current modes.

`sayback(ch) char ch;`
 Prints as many backspaces as necessary to obliterate the full printing of character "ch" under current modes.

`ttoutdriv()`
 Called at interrupt level. Waits a standard delay to slow down the terminal and then sends one character from the output buffer to the terminal. Line feeds are followed by carriage returns. Returns with no action if "paused" is true.

`ttyflush()`
 Removes any character waiting in the terminal input buffer.

`inflush()`
 Clears out the entire input buffer.

`outflush()`
 Clears out the entire output buffer.

3. THE COMMAND INTERPRETER

3.1 General

The command interpreter is a FOREGROUND process that executes commands typed at its console. The command interpreter may start another FOREGROUND process, which communicates with the command interpreter to get command-line arguments.

The command interpreter is compiled by executing "makecom-int", which compiles and links together three files to produce "comint". The three source files are: "comint.u", which handles command line parsing, "comutil.u", which contains routines to execute most commands, and "comrun.u", which executes the "run" command.

3.2 Initialization

The command interpreter acquires a file manager link, a terminal driver link, and terminal input and output links from the resource manager. The terminal driver link is only used to request or change console modes; initially, the command interpreter sets these to "ECHO".

3.3 Command line parsing

A line is input with a "readline" call and converted into a null-terminated string. The line is truncated to LINEMAX-1 characters (LINEMAX is 200).

The routine "findargs" scans the input line, separating it into arguments. Sequences of characters enclosed in quotes are left alone, with the quotes deleted. The Boolean variable "quoted" is true during this process. Two consecutive quotes encountered while "quoted" is true are converted into one quote and do not turn off "quoted". When "quoted" is false, a blank or tab is converted into a null to terminate an argument. Any immediately following blanks or tabs are ignored; the Boolean variable "spacing" is true during this process. At the beginning of the line, "quoted" is false and "spacing" is true. The array "argvec" returns pointers to the argument locations; an entry is made in "argvec" when "spacing" changes from true to false. The variable "argcount" tells the number of arguments; it is incremented when "spacing" changes from false to true or at the end of the line if "spacing" is false.

If there are no arguments, no action is taken. Only the first MAXARGS (10) arguments are used; the rest are ignored.

The routine "lookup" searches a list of character strings to find those whose initial segments match a given string argument. The list format is an alphabetically sorted array of character strings alternating with corresponding codes (integers), and with pseudodata sentinels at each end. Two pointers into the table, "low" and "high", start at opposite ends and move toward each other as the argument is scanned. As each character in the argument is examined, "low" moves up the table so long as this character is larger than the corresponding ones in the table at which "low" points; "high" does the reverse. The process stops if the argument is exhausted or if "high" and "low" pass each other. In the latter case, there is no match. In the former, there are one or more matches; "low" and "high" are equal or not accordingly.

The first argument on the command line is deciphered as a command by calling "lookup" with the table "commands". If there is a unique match, the appropriate action is taken.

3.4 Command execution

The "background" command starts a process with modes "BACKGROUND" and "REUSE" and passes the given argument as an integer. An answer is received from the resource manager and the new process's process identifier is printed. The new process is not given a link to the command interpreter.

The "copy" and "type" commands are translated into "run copyfile" commands. The program "copyfile" is an independent program

that copies one file to another, with the terminal as the default for the second file.

The "directory" command executes a "stat" on the indicated file and prints selected portions of the information returned.

The "make" command "creates" the indicated file and performs "read" commands for IOBUFSIZE (100) bytes at a time from the terminal. After each "read", a "write" is done to the file; finally, the file is closed. The end is indicated by a "read" returning less than 512 bytes; thus, if the input has exactly 512 bytes (or a multiple thereof), it must be terminated by an extra control-D.

The "run" command starts a process with modes "BACKGROUND" and "REUSE" and passes as an argument the number of command line arguments. The resource manager is given a REQUEST link for the child and the terminal input link is closed so that the child may open it. Command line arguments are sent to the child when requested. The command interpreter assumes that the child has terminated when the REQUEST link is destroyed; it then reads the next console command. If the start fails, the command interpreter waits for the REQUEST link to be destroyed before continuing.

The "set" or "SET" command first requests the current modes, which causes the terminal driver to print them. The command line arguments are then deciphered individually; a "-" prefix is remembered with the Boolean variable "notflag" and the command itself is decoded by calling "lookup" (Section 3.3) with the table "modetab". When a mode is recognized, the current mode

specification is altered accordingly. Finally, the modes are changed, and the terminal driver prints the new modes.

The "time" command, if given an argument, sets the time by calling "datetol" and "setdate". The argument is only checked to see that it has ten characters, and zeroes are added for the number of seconds. With or without an argument, "time" finally prints the current time, which is done by calling "date" and "ltodate".

Files

comint.u, comutil.u, comrun.u, comint.h

Data Structures

char *commands[]
Holds the known commands paired with an internal distinguishing code. The array must be in alphabetical order.

char *argvec[MAXARGS]
The arguments to started processes are stored here.

char *modetab[]
A table of terminal mode names to be used with the routine "lookup".

Procedures

main()
Initializes tables, acquires file manager and terminal driver links, then executes a loop that accepts commands from the terminal and dispatches them.

int findargs(line) char *line;
"Line" is null-terminated (without final newline) and doesn't contain any embedded nulls. Puts pointers to the beginnings of arguments into the array "argvec" and the count of how many were found into the global "argcount". Terminates the arguments with nulls. Spaces and tabs are considered delimiters unless they appear in quotes ("). Two consecutive quotes inside quotes are considered one quote. Other quotes are stripped.

lookup(str,table,tablesize,result) char *str, **table; int *result;
Looks up character string "str" in "table". Sets result[0] and result[1] such that table[result[0]], table[result[0]+1], ... , table[result[1]] all have "str" as an initial segment. If result[0]>result[1], there was no match. Assumes that table[0], table[2], ... are strings kept sorted in alphabetical order, and table[1], table[3], ... are other data to be ignored in lookup. Assumes further that table[0] is guaranteed to compare low and table[tablesize-2] is guaranteed to compare high with "str".

```

int intype(fname) char *fname;
    Handles a "make" command. Accepts input from the terminal,
    creates a new file with name "fname" and puts all input on
    that file. Returns 0 on success, -1 on failure.
dir(fname) char *fname;
    Handles a "dir" command. Uses the file manager to read the
    directory information from a file, and prints it on the ter-
    minal.
setmodes()
    Handles a "set" command. Uses "lookup" to find what modes
    are requested, and communicates with the terminal driver to
    establish those modes.
prnttime()
    Handles a "time" command. Finds the current time with the
    service call "date" and the library routine "ltodat" then
    prints the result.
settime(s) char *s;
    Handles a "time" command with an argument. Uses the library
    routine "datetol" and the service call "setdate" to change
    the kernel's date.
int runback(fname,arg) char *fname;
    Attempts to run the file "fname" as a background process,
    handling the "back" command. It returns the process id of
    the new process or -1 on failure.
killback(procid)
    Sends a note to the resource manager to kill the process
    whose identifier is "procid". Handles the "kill" command.
int run(fname,argc,arg0) char *fname;
    Handles the "run" command. Attempts to load and run the ex-
    ecutable file named by "file". Returns 0 on success. If
    "argc" > 0 then uses "arg0" and following arguments to
    satisfy requests for arguments instead of arguments from the
    command line. Executes a loop that waits for requests from
    the child for arguments until the child terminates.

```

4. THE FILE MANAGER

4.1 General

The file manager forwards requests from other processes to the demon running on the PDP-11/40 where they are implemented under Unix. (See Section 5 for details on the demon.)

The code lies in "filesys.u"; all processes using the file manager should include "filesys.h". (It isn't necessary to in-

clude "filesystem.h" if all communication is done by library routines.)

The word-parallel line used to communicate with the PDP-11/40 uses three registers at location DR11.40 (octal 167770). All reading and writing use busy waits. More details are found in the file "io.h".

The file manager initially gives a REQUEST link to its parent (usually the resource manager) with channel 1. All "open", "create", "alias", "unlink", and "stat" requests come over this link or copies thereof. When a file is "opened" or "created", a new link with channel 2 is enclosed with the reply. This link will be used for "read", "readline", "write", and "seek" requests; its destruction indicates a "close" request.

4.2 Execution of requests

The file manager executes most requests by receiving a message from the client, writing a request over the word-parallel line to the PDP-11/40, reading the reply from the word-parallel line, and sending it to the client.

Requests on channel 1 contain file names. These are communicated over the word-parallel line by first writing the length and then the name. The routine "rawopcr" is used by "open", "create", "alias", and "unlink" to send a request to the demon and receive a one-word reply to be forwarded to the client. In the cases of successful "open" or "create" calls, a new link with channel 2 is enclosed with the reply; the code for this link is the same as the value returned to the client (a Unix file

descriptor). This new link is of type GIVEALL but not DUPALL. The routine "rawstat" is used by "stat"; it reads 38 bytes from the demon. The first word tells whether the stat was successful; either 0 or 36 bytes are forwarded to the client accordingly.

A request on channel 2 refers to an open file; the file descriptor for this file is the code of the link. The routine "rawread" forwards a "read" or "readline" request to the demon and reads the reply. An integer telling how many bytes were actually read comes first, followed by the bytes themselves. The bytes read are then forwarded to the client. No more than MSLEN bytes should be read at a time, so that one message suffices for the reply. The routine "rawseek" similarly treats "seek" requests, except that only one word is read from the demon, and then forwarded to the client.

When a file is opened for writing, the corresponding entry in the array "bytesleft" is set to zero. ("Bytesleft" is indexed by file descriptors.) When a "write" request is received on channel 2, the indicated length for the write is inserted in "bytesleft". Further messages on the same link are taken as data to be written (MSLEN bytes at a time) until the write is completed. As each portion is received, the routine "rawwrite" sends it to the demon and waits for an acknowledgment before proceeding. No reply is given to the client.

When a link on channel 2 is destroyed, a "close" message is sent to the demon. No response is read from the demon, and no reply is made to the client.

Files

filesys.h, filesys.u, io.h

Procedures

main()

Initializes tables, then executes a loop awaiting client requests and dispatching them.

rawstat(name,replylink) char *name;

Handles a "stat" request. The file "name" is sent to the demon. Its answer is returned to the client; failure is marked by an empty message.

int rawopcr(file,mode,how) char *file;

The argument "how" is OPEN, ALIAS, UNLINK, or CREAT. A message is sent to the demon to do the appropriate action to "file". The "mode" is the same as Unix mode for files. The file descriptor given by the demon is returned.

rawread(rwfd,buf,bytes) char *buf;

Reads "bytes" bytes from the file whose descriptor is "rwfd" into "buf", which must be on a word boundary (even).

readr(buf,rwlen) int *buf;

Reads ceiling(rwlen/2) words from the DR line to Unix into "buf", which must be word-aligned (even).

writedr(buf,rwlen) int *buf;

Writes ceiling(rwlen/2) words to the DR line to Unix from "buf", which must be word-aligned (even).

rawclose(usrcode)

Handles a client "close" request. Sends a note to the demon to close the file whose descriptor is "usrcode".

rawwrite(rwfd,buf,bytes) char *buf;

Handles a "write" request from a client. Gives the demon data from "buf" of length "bytes" to be placed in file "rwfd".

int rawseek(skfd,offset,mode)

Handles a "seek" request from a client. Sends a note to the demon to do the given seek ("offset" and "mode" mean the same as in Unix) to file identified as "skfd". Success returns 0; failure -1.

5. THE DEMON

5.1 General

The "demon" is a program that runs on the PDP-11/40 under Unix. Its code is in "demon.c". Roscoe processes that communicate with it must include "demon.h".

For each LSI there is an associated demon. This demon reads from a word-parallel line connected to that LSI; the Unix names

for these lines are `"/dev/drx"`, where $x = 0, \dots, 4$. Commands are translated into corresponding Unix system calls and appropriate responses are written to the word-parallel line. All user process communication at the LSI side of the fence is done by the file manager (Section 4).

Each message sent in either direction on the word-parallel line is preceded by at least one header word of "NONSENSE" (octal 125252). After the header word(s), the next three words of a message to the demon have the following structure:

```
struct {
    int command, code, length;
}
```

The number of bytes remaining in the message is "length". These remaining bytes are usually a file name, in which case they will subsequently be read into the character array "fname", of size MAXNAME (40). The value returned over the word-parallel line is usually a single word (after a word of "NONSENSE").

The demon sits in an infinite loop awaiting messages. When a message is received, the appropriate action is taken, as described in further subsections. The routine "getstr" is used to read from the word-parallel line; it rounds the number of bytes up to an even integer and watches out for errors due to terminal interrupts. The routine "signal" is called to catch terminal interrupts, which otherwise plague all Unix processes started at a given terminal.

5.2 DALIAS command

The string "fname" is split into two pieces to become the two arguments for the Unix call "link". The length of the first substring is "code". The effect of "link" is to make its second argument an alias for the first one. The value returned by "link" is passed on.

5.3 DCLOSE command

file descriptor "code" is closed (Unix call "close"). No message is returned.

5.4 DCREAT command

The file "fname" is created (Unix call "creat") with mode "code". The value returned by "creat" is passed on.

5.5 DOPEN command

The file "fname" is opened (Unix call "open") with mode "code". The value returned by "open" is passed on.

5.6 DREAD command

For this command, "length" tells the number of bytes to read from file descriptor "code", using the Unix call "read". This length is truncated to "BUFLen" (512). The first word of the return message is "code". The second word is the value returned by "read", which tells the number of bytes actually read. The bytes read are written next; if "read" returns -1 (error), nothing else

is written. A garbage byte will exist at the end if the number of bytes actually read was odd.

5.7 DREADLINE command

This command is identical to "DREAD", except that the Unix call "read" is used for one byte at a time. If a "newline" character is encountered, it is considered part of the returned text, and reading stops.

5.8 DSEEK command

A Unix call "seek" is performed on file descriptor "code". The offset for the "seek" call is "length"; the mode for the "seek" call is the next word read from the word-parallel line. The value returned by "seek" is passed on.

5.9 DSTAT command

A Unix call "stat" is performed on file "fname". The first word of the return message is -1 for failure, 36 for success. In either case, 36 additional bytes are written; if the "stat" succeeded, these bytes are the desired information.

5.10 DTIME command

A Unix call "time" is performed to return a double word. The first word of the return message is 0; the next two words are the result of the "time" call. This command is only used by the resource manager during Roscoe initialization.

5.11 DUNLINK command

A Unix call "unlink" is performed on the file "fname". The value returned by "unlink" is passed on.

5.12 DWRITE command

The rest of the incoming message is read into "writebuf"; the length of this text is "length", truncated to BUFLen (512) bytes. This text is then written to file descriptor "code", using the Unix call "write". The value returned is "code" if "write" returned success; otherwise, the value returned is "code" times minus one.

6. LIBRARY ROUTINES

All the files in this section are in the directory "/usr/network/roscoe/library". The object code is archived in "libr.a".

6.1 File manager routines

These routines communicate with the file manager (Section 4).

The routine "opcreat" is used by "open", "create", "alias", "unlink", and "stat" (the sources reside in "opcr.u", "open.u", "create.u", "alias.u", "unlnk.u", and "stat.u", respectively) to send a command and file name to the file manager over the given file manager link. Another argument, "mode", has various meanings for "open", "create", and "alias" calls. In the case of

"stat", an additional argument represents a REPLY link that is passed to the file manager. The routine "stat" receives a response over this link and copies the information into the designated buffer. In the other four cases, "opcreat" waits for a response from the file manager and gives a return value accordingly (this value is a link number after a successful "open" or "create"). The routine "alias" concatenates its two file name arguments before calling "opcreat"; "mode" is then the length of the first name, to eventually be decoded by the demon.

The routine "close" (in "close.u") is synonymous with "destroy".

The routine "someread" (in "somerd.u") is used by both "read" and "readline" (in "read.u" and "rdln.u", respectively). The appropriate command is sent over the given link (to either the file manager or terminal driver). All reads are split up into individual requests for MSLEN bytes at a time. The responses for the portions of the read are copied into the given buffer.

The routine "seek" (in "seek.u") sends an appropriate message to the file manager over the given link, and waits for a reply. Success is reported by a zero in the first word of the reply message.

The routine "write" (in "write.u") sends the appropriate header message over the given link (to the file manager or terminal driver) and then sends the data in subsequent messages MSLEN bytes at a time. No reply is awaited, and no value is returned. The routine "print" (in "print.u") is similar to the Unix printf routine. It edits the output string and calls "write". A linear

buffer, "prbuf", of size PRINTBUFSIZE (100), is used. The format string is scanned and the routines "printint", "printlong", "printoct", and "printstr" are called to handle the conversions for "%d", "%w", "%o", and "%s" format items, respectively. Both "printint" and "printlong" check for the sign, take absolute value, and use division by 10 (recursively), although "printlong" uses long arithmetic. The routine "printoct" always produces six characters; it first checks the sign bit, and then inspects three bits at a time with an appropriate shift. In all cases "printch" is used to put characters into "prbuf" and to call "write" when the buffer is full. The buffer is also flushed at the end.

6.2 Resource manager request routines

The routines "fsline" and "parline" (in "fsline.u" and "parlin.u") ask the resource manager (Section 1) for the appropriate link and return the enclosure. The routines "inline" and "outline" (in "inline.u" and "outlin.u") first ask the resource manager for a terminal link, then ask the terminal driver for the appropriate line, and finally return the enclosure.

The routine "fork" (in "fork.u") sends a start message to the resource manager, conveying the file name, argument, and mode, but always specifying ANSWER. A link is given to the child of type REQUEST, GIVEALL, and TELLDEST. The first word of the reply message is returned; in particular, this word is the process identifier for a BACKGROUND child. If the start failed, "fork" waits for the given link to be destroyed.

The routine "killoff" (in "killof.u") conveys the kill request to the resource manager, gets a reply, and returns the first word of the reply message.

6.3 Roscoe service calls

The Roscoe service call interface is the assembler file "lib.s". For each call, an appropriate magic number is placed in register 1 and a jump is made to the Roscoe entry point "sys" (octal location 1002). Arguments are left on the stack; the kernel takes it from there.

6.4 Miscellaneous

The routine "atoi" (in "atoi.u") converts a string into an integer.

The file "call.u" contains "call" and "recall". The routine "call" sends a message as indicated, encloses a REPLY link, puts the REPLY link's code into the global variable "unique", and invokes "recall". The latter receives a message with a five second delay, and checks that the incoming message has the proper code ("unique") and note ("DATA").

The assembler file "reset.s" contains "setexit" and "reset". The routine "setexit" saves register 5 and the old program counter in global locations "sr5" and "spc". The routine "reset", by restoring these, effects a return to the environment which last called "setexit".

The file "user.h" contains various macros freely referred to in this documentation. For the user's convenience, it also de-

defines TRUE (all 1's) and FALSE (all 0's) and the following structures:

```
struct {char lowbyte,highbyte;};
struct {int word1,word2;};
```

The file "time.u" contains the routines "datetol" and "lto-date", which convert character strings into long integers (representing seconds since the beginning of 1973) and vice versa, respectively. The array "calendar" contains the number of days preceding each month in a leap year, with pseudodata "366" as a thirteenth entry. Character string arrays store the days of the week and months of the year. The macro FOURYEARS gives the number of days in a four-year period. The first step of "datetol" is to convert the given string, with format "yymmddhhmmss" into an array of six two-digit integers. Arrays "lbound" and "ubound" are used to check that these integers are reasonable. Sizes of months are also checked by subtracting the appropriate consecutive entries in "calendar". The number of days is calculated by computing the number of four-year intervals beginning with 1973 (and multiplying by FOURYEARS), then adding on the proper (0-3) number of (non-leap) years (times 365), then adding on the month offset as found in "calendar", and finally adding in the day of the month. For non-leap years, February 29th is caught as a mistake, and any day occurring later in the year is decreased by one. Finally, hours, minutes, and seconds are added on. The reverse process is carried out by "ltodate". Seconds, minutes, and hours are first removed. The day of the week is computed from the number of days modulo 7. Division by FOURYEARS determines the four-year period; the remainder determines the ex-

act year and day within the year, with a remainder of (4*365) representing December 31st of a leap year. In a non-leap year, conversion (to the proper format for "calendar") is performed by increasing by one any day of the year larger than 58. (February 28th remains 58; March 1st is bumped to 60; etc.) The month is calculated by dividing the number of days by 30; the answer may be too large by one and is corrected by inspecting "calendar". As the result is computed, it is edited into a character string.

REFERENCES

- Finkel, R. A., Solomon, M. H., The Roscoe Kernel, University of Wisconsin -- Madison Computer Sciences Technical Report #337, September 1978.
- Solomon, M. H., Finkel, R. A., ROSCOE -- a multiminicomputer operating system, University of Wisconsin -- Madison Computer Sciences Technical Report #321, September 1978.
- Tischler, R. L., Solomon, M. H., Finkel, R. A., ROSCOE User Guide, University of Wisconsin -- Madison Computer Sciences Technical Report #336, September 1978.

APPENDIX H

**Arachne User Guide
Version 1.2**

*Raphael Finkel
Marvin Solomon
Ron Tischler*

**Technical Summary Report 2066
Mathematics Research Center
University of Wisconsin-Madison
April 1980**

UNIVERSITY OF WISCONSIN-MADISON
MATHEMATICS RESEARCH CENTER

ARACHNE USER GUIDE[†]

Version 1.2

Raphael Finkel, Marvin Solomon and Ron Tischler

Technical Summary Report #2066
April 1980

ABSTRACT

Arachne is a multi-computer operating system running on a network of LSI-11 computers at the University of Wisconsin. This document describes Arachne from the viewpoint of a user or a writer of user-level programs. All system service calls and library routines are described in detail. In addition, the command-line interpreter and terminal input conventions are discussed. Companion reports describe the purposes and concepts underlying the Arachne project and give detailed accounts of the Arachne utility kernel and utility processes.

AMS (MOS) Subject Classifications - 68-00, 68A35, 68A45, 68A55

Key Words - Distributed computing, Networks, Operating systems,
User interface

Work Unit Number 3 (Numerical Analysis and Computer Science)

[†] Appeared as Computer Sciences Technical Report 379, Computer Sciences Department, University of Wisconsin-Madison. We have been forced to change the name of the Roscoe distributed operating system, since Roscoe is a registered trademark of Applied Data Research, Incorporated. The new name we have chosen is Arachne; the operating system and research continue unchanged.

SIGNIFICANCE AND EXPLANATION

Arachne is an experimental operating system for controlling a network of microcomputers. It is currently implemented at the University of Wisconsin on a network of five minicomputers. Some of its essential features are: All processors are identical, although they may differ in peripheral units. No memory is shared between processors, and all communication involves messages passed between processes. The way in which the processors are interconnected is not important. The network appears to the user to be a single machine.

This report describes Arachne from the viewpoint of a user or a writer of user-level programs.

The responsibility for the wording and views expressed in this descriptive summary lies with MRC, and not with the authors of this report.

TABLE OF CONTENTS

1.	INTRODUCTION.....	1
1.1	Purpose of this Document.....	2
1.2	Caveat.....	3
1.3	Format of this Guide.....	3
1.4	Revisions.....	4
2.	ROSCOE CONCEPTS AND FACILITIES.....	5
2.1	Processes.....	6
2.2	Links.....	7
2.3	Messages.....	8
2.4	Link restrictions.....	8
2.5	Service calls.....	9
2.6	Utility processes.....	9
2.7	Library routines.....	11
3.	SUBJECT-AREA GUIDE.....	11
3.1	Links and Messages.....	11
3.2	Processes.....	13
3.3	Timing.....	14
3.4	Interrupts and Exceptions.....	15
3.5	Input/Output.....	16
3.6	Miscellaneous Routines.....	17
3.7	Preparing User Programs.....	17
4.	— ROSCOE PROGRAMMER'S MANUAL.....	18
4.1	Alias (Library Routine).....	18
4.2	Awaken (Service Call).....	18
4.3	Call (Library Routine).....	19
4.4	Catch (Service Call).....	20
4.5	Close (Library Routine).....	21
4.6	Copy (Service Call).....	22
4.7	Create (Library Routine).....	22
4.8	Date (Service Call).....	23
4.9	Datetol (Library Routine).....	23
4.10	Destroy (Service Call).....	23
4.11	Die (Service Call).....	23
4.12	Display (Service Call).....	24
4.13	Errhandler (Service Call).....	24
4.14	Fork (Library Routine).....	25
4.15	Fsline (Library Routine).....	26
4.16	Handler (Service Call).....	26
4.17	Inline (Library Routine).....	27
4.18	Kill (Service Call).....	27
4.19	Killoff (Library Routine).....	27

4.20	Link (Service Call).....	28
4.21	Linkok (Service Call).....	29
4.22	Load (Service Call).....	30
4.23	Ltodate (Library Routine).....	31
4.24	Nice (Service Call).....	31
4.25	Open (Library Routine).....	31
4.26	Outline (Library Routine).....	32
4.27	Parline (Library Routine).....	32
4.28	Print (Library Routine).....	32
4.29	Read (Library Routine).....	33
4.30	Readline (Library Routine).....	33
4.31	Recall (Library Routine).....	34
4.32	Receive (Service Call).....	34
4.33	Remove (Service Call).....	36
4.34	Seek (Library Routine).....	36
4.35	Send (Service call).....	36
4.36	Setdate (Service Call).....	37
4.37	Startup (Service Call).....	38
4.38	Stat (Library Routine).....	39
4.39	Time (Service Call).....	39
4.40	Unlink (Library Routine).....	40
4.41	Write (Library Routine).....	40
5.	CONSOLE COMMANDS.....	40
5.1	alias <filename1> <filename2>.....	41
5.2	background <filename> <arg>.....	41
5.3	copy <filename1> <filename2>.....	42
5.4	delete <filename>.....	42
5.5	dump <address>.....	42
5.6	help.....	42
5.7	kill <arg>.....	42
5.8	make <filename>.....	42
5.9	rename <oldname> <newname>.....	43
5.10	run <filename> { <arg> } { ^ <filename> { <arg> } }.	43
5.11	set <modelist> or SET <modelist>.....	43
5.12	time <format>.....	44
5.13	type <filename>.....	44
6.	TERMINAL INPUT PROTOCOLS.....	44
7.	UTILITY PROCESS PROTOCOLS.....	45
7.1	Input/Output Protocols.....	46
7.2	Resource Manager Protocols.....	48
8.	ACKNOWLEDGEMENTS.....	52
9.	REFERENCES.....	53

ARACHNE USER GUIDE⁺
Version 1.2

Raphael Finkel, Marvin Solomon and Ron Tischler

1. INTRODUCTION

Arachne is an experimental operating system for controlling a network of microcomputers. It is currently implemented on a network of five Digital Equipment Corporation LSI-11 computers connected by medium-speed lines.* The essential features of Roscoe are:

1. All processors are identical. Similarly, all processors run the same operating system kernel. However, they may differ in the peripheral units connected to them.

2. No memory is shared between processors. All communication involves messages explicitly passed between physically connected processors.

3. No assumptions are made about the topology of interconnection except that the network is connected (that is, there is a path between each pair of processors). The connecting hardware is assumed to be sufficiently fast that concurrent processes can cooperate in performing tasks.

⁺ Appeared as Computer Sciences Technical Report 379, Computer Sciences Department, University of Wisconsin-Madison. We have been forced to change the name of the Poscoe distributed operating system, since Roscoe is a registered trademark of Applied Data Research, Incorporated. The new name we have chosen is Arachne; the operating system and research continue unchanged.

* This equipment was purchased with funds from National Science Foundation Research Grant #MCS77-08968.

Sponsored by the United States Army under Contract Nos. DAAG29-75-C-0024 and DAAG29-80-C-0041.

4. The network appears to the user to be a single powerful machine. A process runs on one machine, but communicating processes have no need to know if they are on the same processor and no way of finding out. (Migration of processes to improve performance is transparent to the processes involved.)

5. The network is constructed entirely from hardware components commercially available at the time of construction (January, 1978).

6. The software is all functional. Although Roscoe has undergone much revision, it has been working for over a year.

1.1 Purpose of this Document

This document describes Arachne from the point of view of a user or user-programmer. It is both a tutorial and a reference guide to the facilities provided to the user. All information necessary to the programmer of applications programs should be found here.

Further discussion of the concepts and goals of Arachne are discussed in [Solomon 78, 79]. That document also lists some research problems that the Arachne project intends to investigate. The operating system kernel that provides the facilities listed below is described in considerable detail in [Finkel 78, 80b]. Similar detailed documentation about utility processes (such as the File System Process, the Teletype Driver, the Command Interpreter, and the Resource Manager) is contained in [Finkel 79a, 79b].

Arachne has been developed with extensive use of the UNIX

operating system [Ritchie 74]. All code (with the exception of a small amount of assembly language) is written in the C programming language [Kernighan 78]. The reader of this document is assumed to be familiar with both UNIX and C.

A new programming language called Elmer, is being designed for applications programs under Arachne; it will be described in a future report. Arachne programs may be written in either Elmer or C. Currently, the library is available only in C.

1.2 Caveat

Arachne is in a state of rapid flux. Therefore, many of the details described in this Guide are likely to change. The reader who intends to write Arachne programs should check with one of the authors of this report for updates.

1.3 Format of this Guide

Section 2 provides an overview of the concepts and facilities of Arachne. Section 3 describes the facilities by name, arranged according to general subject areas. Section 4 is a programmer's reference manual. Each function is listed alphabetically, its syntax and purpose are described, and it is classified as a service call (an invocation of an operating system kernel routine) or a library routine (a procedure linked into the user program). Section 5 describes the command line interpreter and lists the commands that may be entered from the terminal. Section 6 describes the conventions governing terminal input/output. Section 7 presents protocols for communicating

with the various utility processes.

1.4 Revisions

The following changes have been made to Arachne since version 1.0 of this document:

There is a new service call, "linkok", to determine if a link number is currently valid. The library routine "call" uses this service call to avoid sending a message across a bad link.

Messages now include length information. The library routines "call" and "recall" have been modified to reflect this change. The file and terminal protocols have also been simplified.

The following changes have been made since version 1.1 of this document:

A new utility process, the pipe, is now available. Pipes allow the output of one user process to be attached to the input of another.

The structure "usmesg" has been abolished, and "urmesg" no longer contains the body of the message. Instead, both "send" and "receive" have a new argument that specifies the message body.

A new link restriction, MAYERROR, is orthogonal to all other restrictions. The last argument to send may have the ERROR bit on, in which case the message is considered an error report if the link across which it is sent has MAYERROR specified. Receipt of an error report raises an exception.

The "die" service call now takes a character-string argu-

ment. This argument becomes the body of any DESTROYED message that is generated due to the termination of the calling process.

When a process dies, error reports are sent along any links that it holds with restriction MAYERROR but not TELLDST.

Many errors caused by service calls raise exceptions. An exception can only occur during a service call. If it is not caught, the guilty process terminates. Exceptions may be caught with the "errhandler" service call.

A new facility for asynchronous message receipt, called "catch", allows a procedure to be specified that will be invoked as soon as a message arrives on the specified channels.

The "display" kernel call returns timing information about the owner of any link.

We have been forced to change the name of the Roscoe distributed operating system, since Roscoe is a registered trademark of Applied Data Research, Incorporated. The new name we have chosen is Arachne; the operating system and research continue unchanged.

2. ROSCOE CONCEPTS AND FACILITIES

The fundamental entities in Arachne are: files, programs, core images, processes, links, and messages. The first four of these are roughly equivalent to similar concepts in other operating systems; the concepts of links and messages are idiomatic to Arachne. A file is a sequence of characters on disk. Each file has directory information giving the time of last modification and restrictions on reading, writing, and execution. The con-

tents of a file may contain header information that further identifies it as an executable program. Version 1 of Arachne uses the UNIX file system; therefore, the reader familiar with UNIX should have no problem understanding Arachne files.

Program files contain text (machine instructions), initialized data, and a specification of the size of the uninitialized global data space (bss) required by the program. Program files also contain relocation information and an optional symbol table.

2.1 Processes

A process is a locus of activity executing a program. Each process is associated with a local data area called its stack. A program that never modifies its global initialized or bss data but only its local (stack) data is re-entrant, and may be shared by several processes without conflict. A main-storage area containing the text of a program, its initialized data, and a bss data area, but not including a stack, is called a core image. Core images may not share text areas unless they are reentrant; the text and data areas of Elmer programs are loaded separately, so Elmer programs may share text even if they are not reentrant. The initiation of a process entails locating or creating (by loading) a core image, allocating a stack, and initializing the necessary tables to record its state of execution. Similarly, when a process dies, its tables are finalized and its stack space is reclaimed. If no other processes are executing in its core image, then the space occupied by the core image is available for re-use.

2.2 Links

All communication is performed by message passing across links. A link combines the concepts of a communications path and a "capability." A link represents a logical one-way connection between two processes, and should not be confused with a line, which is a physical connection between two processors. The link concept is central to Arachne. It is inspired and heavily influenced by the concept of the same name in the Demos operating system for the Cray-1 computer [Baskett 77]. Each link connects two processes: the holder, which may send messages over the link, and the owner, which receives them. The holder may duplicate the link or give it to another process, subject to restrictions associated with the link itself. (See "Link restrictions" below.) The owner of a link, on the other hand, never changes.

Links are created by their owners. When a link is created, the creator specifies a code and a channel. The kernel automatically tags each incoming message with the code and channel of the link over which it was sent. Channels are used by a process to partition the links it owns into subsets: When a process wants to receive a message, it specifies a set of channels. Only a message coming over a link corresponding to one of the specified channels is eligible for reception. A link is named by its holder by a small positive integer called a link number, which is an index into a table of currently-held links maintained by the kernel for the holder. All information about a link is stored in this table. (No information about a link is stored in the tables

of the owner.)

2.3 Messages

A message may be sent by the holder to the owner of a link.

A message may contain, in addition to MSLEN (currently 40) characters of text, an enclosed link. The sender of the message specifies the link number of a link it currently holds. The kernel adds an entry to the link table of the destination process and gives its link number to the recipient of the message. In this way, the recipient becomes the holder of the enclosed link. If the original link is not destroyed, the sender and the recipient hold identical copies of the link.

2.4 Link restrictions

Links may be created with various restrictions. These can be characterized as modes, permissions, and notifications. The orthogonal modes are REQUEST and REPLY. A reply link is distinguished by the fact that it can only be used once; it is destroyed when a message is sent over it. A reply link may not be the enclosed link in a message sent over another reply link. Similarly, a request link cannot be sent over a request link. These restrictions enforce a communication protocol in which all communications between two processes connected by a REQUEST link are initiated by the holder of that link.

Two permissions are GIVEALL and DUPALL, controlling distribution of the affected link to other parties. A third permission is MAYERROR, which allows the holder to send an error message,

whose receipt will raise an exception.

The notifications are TELLGIVE, TELLDUP, and TELLEDEST. When these restrictions are in force, unforgeable messages are sent to the owner of the link when it is given away, duplicated, or destroyed. (The last of these messages contains a body provided by the holder if it dies holding the link.)

2.5 Service calls

The Arachne kernel is a module that resides identically on all the machines of the network and provides various services for user programs. The services are requested by means of service calls, which appear to the caller to be procedure invocations.

The chief function of the kernel is to support link maintenance and message passing by providing service calls to create and destroy links and send, receive and catch messages. Additional service calls create and destroy processes, read and set "wall-clock" and high-resolution interval timers, specify a handler to catch exceptions, and establish interrupt handlers for processes that control peripheral devices.

2.6 Utility processes

Arachne has been designed so that as many as possible of the traditional operating system functions are provided not by the kernel, but by ordinary processes. These utility processes may invoke service calls not intended to be used by the casual user, but otherwise they behave exactly like user processes. The terminal driver is an example. One terminal driver resides on each

processor that has a terminal. All terminal input/output by other processes is requested by messages to this process. It understands and responds to most commands accepted by a file (see below), as well as a few extra ones, such as "set modes" (e.g., echo/no echo, hard copy/soft copy).

A file manager process has access to the Arachne file system, currently implemented on the supporting PDP-11/40. A request to open a file sent to any file manager process causes a link to be created representing the open file. To the user of a file, the open file behaves like a process that understands and responds to messages requesting read and write operations. The file is closed by destroying the link. A version of the file manager that uses a floppy disk instead of the PDP-11/40 file system is also available; it follows the same protocols as the other file manager.

The most complex utility process is the resource manager (RM). Resource managers reside on all processors and are connected by a network of links. A process can request an RM to create a new process. The RM may create the process on its own machine or relay the request to another RM based on such considerations as location of the process that requested the creation, availability of free memory, proximity of resources such as devices and files, and the possibility that the required program is already in memory.

The new process is started with a link to its RM, over which it can request links to the process that requested its creation, to a file manager process, to a terminal driver, or to other

resources. The RM can kill the process, or it can give a special link to another process (usually a terminal driver) that may be used to kill it.

2.7 Library routines

Functions provided by service calls are rather primitive, and communication with utility processes can involve complicated protocols. An extensive library of routines has been provided to simplify writing of programs that use service calls and utility processes. These routines serve to hide the communication necessary to accomplish various tasks, and make it especially easy to introduce software not originally designed for the Arachne environment. These routines can only be used with C programs; the Elmer library is under construction.

3. SUBJECT-AREA GUIDE

This section lists service calls and library routines by subject area.

3.1 Links and Messages

A new link is created by a process through the "link" service call. Initially, the creator is both holder and owner of the link. The creator specifies what channel and code to associate with the link, so that future messages arriving along it can be selectively received and identified. In addition, the creator may place restrictions on the use of the link, controlling wheth-

er or not it may be given to third parties, duplicated, or used repeatedly, and requiring notifications to be sent along it in the event of link duplication, transferral, or destruction. Finally, links may specify that they can carry error messages. Receipt of an error message terminates the recipient.

Messages are sent with the "send" service call, which specifies a link over which the message is to be sent, the message text and an optional enclosed link. It also indicates if the message is an error message.

Messages are accepted by "receive," which specifies a set of channels, a place to put the message, and a maximum time the recipient is willing to wait. "Receive" can also be used to sleep a specified period of time by waiting for a message that will never arrive. Asynchronous message receipt is accomplished by "catch", which has the same arguments as receive, except it has no wait time, and it specifies a procedure to call when an appropriate message arrives. This catcher procedure is very limited in the kernel calls it can perform.

A simple send-receive protocol is embodied in the library functions "call" and "recall," which are simpler to use than send and receive, and should be adequate for most routine communication. The "call" library routine sends a message along a given link, enclosing a reply link. It then waits five seconds for a response, which it returns to the caller. If no answer has arrived in five seconds, it returns failure, and the "recall" routine can be invoked to continue waiting for the tardy response.

3.2 Processes

A process may spawn others by communicating with the resource manager; typical cases are handled by the library routine "fork". The requestor indicates whether the child should be run as a foreground, background, or detached job. Foreground processes are attached to a terminal and can be terminated by entry of a control-C. Background processes may only be terminated by requesting the resource manager to remove them, which is accomplished by the library routine "killoff". Detached processes cannot be terminated except at their own request. The caller also indicates whether the child process may share its core image with other processes, whether an old and inactive core image may be used, or whether a fresh core image is required.

Every user process is started holding link number 0, whose destination is the resource manager on that process's machine. When calling "fork", the parent may indicate a link that it wishes to give to the child; the child obtains this link with the library routine "parline", which communicates with the resource manager along link 0. A process can terminate itself by calling "die"; it can yield the CPU to another process by the service call "nice". (Scheduling is not pre-emptive.)

Four low-level process-control service calls are provided for the use of the resource manager; they are not intended for the typical user. The service call "load" arranges for bringing new core images into the processor on which the caller resides. If there is no room, the call returns failure, and the resource

manager can try to find a neighboring resource manager that might have better luck. Once a core image is loaded, processes can be started in it with the service call "startup", which provides the new process with an initial link 0 of the caller's choosing. The "kill" service call removes a process, and "remove" reclaims its core image. The separation of images and processes allows one core image to be used simultaneously by several processes, and a core image may be saved after the last process is gone to speed up the next invocation of a process that would use it.

3.3 Timing

Arachne has two notions of time. One is the wall clock, which keeps track of seconds in real time. Messages sent between resource managers are routinely used to keep the various machines synchronized. There is also an interval timer, which may be used to monitor elapsed time in increments of ten-thousandths of seconds. No process may change the interval timer.

The wall clock is referenced, changed, enciphered, and deciphered by "date", "setdate", "datetol", and "ltodate", respectively. The interval timer is referenced by "time". The percentage of time used by any process may be discovered with "display".

3.4 Interrupts and Exceptions

User programs may handle their own interrupts. This feature is currently used by the terminal driver. A process may establish an interrupt-level routine with the "handler" service call. This call names not only the interrupt handling routine and which interrupt it is intended to service, but also a channel along which to receive messages sent by that interrupt routine. The interrupt-level routine should, of course, be thoroughly debugged and fast. Interrupt-level routines may notify the process that established them by the service call "awaken". This call causes a special message to be sent to the master routine along the channel it specified in its "handler" call. Since the master and interrupt-level routines share code and data, all details of the communication are embedded in shared variables; the awaken message itself is empty.

If a processes arranges for asynchronous receipt of messages by using a "catch" service call (see "Links and Messages" above), then arrival of such a caught message will not preempt any other process. However if the catching process is currently executing, control will immediately switch to the catcher routine within the process.

Exceptions are raised by many service all errors (usually poorly formed service calls) and by receipt of error messages. Usually, exceptions cause the termination of the offending process. Exceptions may be caught by establishing a handler with the "errhandler" service call. When an exception arises, the

Handler will be invoked with arguments indicating the value returned from the failed service call, the service call number, and all the arguments to the service call. Return from the handler acts like return from the service call.

3.5 Input/Output

To use files, a process first obtains a link to the file manager process by calling the library routine "fsline", which communicates with the local resource manager. This link is used in subsequent library routine calls: "open" and "create" make new files or ready old ones for reading or writing, and return links to be used for manipulations of those files. The library routines "read", "write", and "seek" act much like the Unix file primitives of the same name to provide random access into the open file. A file is closed by the library routine "close", which is identical to the service call "destroy", which destroys a link. Finally, the library routine "stat" returns various information about the open file. Each of these library routines packages a request in a message that is sent across the file access link to the file manager process.

To use the terminal, a process obtains input and output links by calling the library routines "inline" and "outline", respectively, which communicate with the local resource manager. An input link can be used to discover or change terminal modes (only the command interpreter uses this feature) and to perform terminal input. An output link can be used for terminal output. These links may also be "closed"; they are closed automatically

when a process dies. The terminal driver allows at most one input link to be open at a time.

Reading is performed by the library routines "read" and "readline". Writing is performed by "write" and, if formatting is desired, by "print". Each of these routines works equally well in dealing with a file instead of the terminal. The service call "printf" is identical to "print" except that it always uses the terminal; it is a debugging tool not intended for the typical user.

The user familiar with UNIX is cautioned against assuming that any particular buffer size is particularly efficient for reads or writes, because Arachne splits up I/O into packets of size MSLEN bytes anyway.

3.6 Miscellaneous Routines

The following routines from the C library also exist in the Arachne library: atoi, long arithmetic routines, reset, setexit, strcpy, streq, strge, strgt, strle, strlen, strt, strne, and substr.

An additional routine supplied by Arachne is "copy".

3.7 Preparing User Programs

User programs for Arachne are written in the C programming language. They are compiled under UNIX on the PDP-11/40 and should include the files "user.h" and "util.h" in directory /usr/network/roscoe/user. Source programs should have filenames ending with ".u". To prepare a file named "foo.u", execute

"makeuser foo", which creates an executable file for Arachne named "foo". The executable files are always stored in /usr/network/roscoe/user.

4. ROSCOE PROGRAMMER'S MANUAL

The following is an alphabetized list of all the Arachne service calls and library routines. For each service call error result, the notation "(*)" indicates that the error causes an exception to be raised.

4.1 Alias (Library Routine)

```
int alias(fslink, fname1, fname2) char *fname1, *fname2;
```

The new name "fname2" is associated with file "fname1". The argument "fslink" is the caller's link to the file manager. The old name is still valid. Possible errors: The combined length of "fname1" and "fname2" must not exceed MSLEN-6. The name "fname2" must not already be in use. File "fname1" must exist. All errors return -1.

4.2 Awaken (Service Call)

```
awaken()
```

Only an interrupt level routine may use this call. It sends a message to the process that performed the corresponding "handler" call along the channel specified by that "handler" call.

Returned values: Success returns a value of 0. -2 is re-

turned if the message cannot be sent because no buffers are available; an "awaken" may succeed later.

4.3 Call (Library Routine)

```
int call(ulink,outmess,inmess,outlen,inlen)
    char *outmess,*inmess; int *inlen;
```

This routine sends a message to another process and receives a reply. The link over which the message is sent is "ulink", which should be a REQUEST link. The argument "outmess" points to the message body to be sent, of size "outlen". (Caution: "outlen" should include the terminating null, if the message is a string.) Similarly, "inmess" points to where the reply body will be put. The length of the reply will be placed in the integer pointed to by "inlen"; if the user doesn't need this feature, "inlen" may be set to 0. If "inmess" is 0, any reply will be discarded. An error is reported if the reply does not arrive in five seconds (see "recall"). In normal cases, the return value is the link enclosed in the return message; it is -1 if there isn't any enclosure. Ignoring errors, the user may consider this routine an abbreviation for:

```
struct urmesg urmess;
send(ulink,link(0,CHAN16,REPLY),outmess,outlen,NODUP);
receive(CHAN16,inmess,&urmess,5);
if (inlen) *inlen = urmess.urlength;
return(urmess.urlnenc);
```

Returned values: Under normal circumstances, the return value is either -1 or a link number. -2 means an error occurred while sending, -3 means the waiting time expired, -4 means that the return link was destroyed, -5 means that something was re-

ceived with the wrong code, meaning that the user program is also using CHAN16 for some other purpose, -6 means that a return link couldn't be created in the first place, -7 means that the ulink was bad.

NOTE: CHAN16 is implicitly used; for this reason, the user is advised to avoid this channel entirely. Several other library routines also invoke "call", and thus use CHAN16.

4.4 Catch (Service Call)

```
int catch(chans,data,urmess,catcher) char *data, int catcher();

struct urmesg {          /* for receiving messages */
    int urcode;           /* chosen by user, see "link" */
    int urnote;           /* filled in by Arachne, see "receive" */
    int urchan;           /* chosen by user, see "link" */
    int urlnenc;          /* index of enclosed link */
    int urlength;         /* length of incoming message */
} *urmess;
```

The arguments are the same as for the receive service call, except for the last one. The procedure specified by "catcher" is activated as an asynchronous message recipient for messages that appear on the channels indicated. If a catcher is active on some channel, then any message that arrives on that channel will cause the asynchronous invocation of the catcher, which takes no arguments. The message itself is placed in "data" and "urmess" in the same way as for "receive".

The catcher procedure may inspect the message and modify global variables; it may not invoke any service calls except "printf". If the catcher returns FALSE, it will be deactivated from the channel across which the message came; if it returns

TRUE, it remains active.

If a catcher has already been activated for some channels, and a new "catch" call names other channels, then the union of all the channels active before and now indicated will be activated for catchers. There is only one catcher procedure, one "data", and one "urmess" at any time; subsequent "catch" calls can replace these values with new ones.

If "catcher" is 0, then instead of activating the given channels, they are deactivated with respect to catching messages. All channels not mentioned in "chans" are unaffected. The "data" and "urmess" arguments are ignored in this case.

If the destination of a message is both waiting to receive it and has a catcher activated to catch it, the message is given to the catcher, not the receive call. Catching a message prevents it from also being received.

Messages are caught in the order in which they arrive at the destination.

Returned values: 0 is returned on success. -1 (*) means the argument "catcher" was bad, -2 (*) means "urmess" or "data" was bad. In this case, the other specified channels may or may not get catchers.

4.5 Close (Library Routine)

int close(file)

The argument "file" is either a link to an open file, or a terminal input or output link. The returned value is 0 on success, negative on failure (actually, "close" is synonymous with

"destroy"). These links are automatically closed when a process dies; however, execution of this command gives the caller more room in its link table. Also, closing the terminal input makes it possible for another process to open it.

4.6 Copy (Service Call)

```
int copy(link);
```

This service call returns a copy of the given link. If the link is restricted, copy may fail or cause a notification to be sent. Returned values: 0 for success, -1 (*) , -2 (*) if the original link number is out of range or not in use, -3 (*) if the link is protected against duplication, -4 (*) if there is no room in the user's link table for a new link. (See "link".)

4.7 Create (Library Routine)

```
int create(fslink,fname,mode) char *fname;
```

If the file named "fname" exists, it is opened for writing and truncated to zero length. If it doesn't exist, it is created and opened for writing. The argument "fslink" is the caller's link to the file manager. The protection bits for the new file are specified by "mode"; these bits have the same meaning as for UNIX files, but all files on Arachne have the same owner. The returned values are as in "open".

4.8 Date (Service Call)

long date();

This service call returns the value of the wall clock, which is a long integer representing the number of seconds since midnight, Jan 1, 1973, CDT.

4.9 Datetol (Library Routine)

long datetol(s) char s[12];

This library routine converts a character array with format "yymmddhhmmss" into a long integer, representing the number of seconds since midnight (00:00:00) Jan 1, 1973. It accepts dates up to 991231235959 (end of 1999); -1 is returned on error.

4.10 Destroy (Service Call)

int destroy(ulink)

Link number "ulink" is removed from the caller's link table.

Returned values: 0 is returned on success. -1 (*) means that the link number is out of range, -2 (*) means that it is an invalid link, and -3 (*) means the link may not be destroyed (link 0 has this property).

4.11 Die (Service Call)

die(mesg) char *mesg;

This call terminates the caller. All links held by the caller are destroyed. As these links are destroyed, DESTROYED messages are sent along all links that have the TELLDEST restric-

tion; these messages contain "mesg" as the body (always MSLEN bytes), unless "mesg" is 0. Error messages are sent along all links that have the MAYERROR restriction but not TELLDEST.

Various errors can cause a "die" to be automatically generated. Here are the possible contents of "mesg":

- bad die message
- bad trap
- exception not caught
- killed
- fell through
- core image damaged

4.12 Display (Service Call)

```
int display(link);
```

This call returns a number in the range 0 to 100 that represents the percentage of CPU time used by the owner of the given link averaged over the last 4 seconds. 0 means that the process has not run at all; 100 means that the process has been active the entire time.

Returned values: -1 (*) if the link points to a different machine, -2 (*) if the link number is invalid.

4.13 Errhandler (Service Call)

```
char *errhandler(addr) char *addr;
```

A new exception handler is established to catch exceptions raised during service calls and receipt of error messages. The handler is a routine at location "addr". The old handler address is returned. A 0 value for "addr" disables exception catching.

If not caught, exceptions cause the termination of the of-

fending process. When an exception arises, the handler will be invoked with these arguments: the value returned from the failed service call, the service call number, and all the arguments to the service call. Return from the handler acts like return from the service call. To ignore exceptions, use a handler that only returns its first argument.

Returned values: -1 (*) if "addr" is unreasonable, the address of the old handler (possibly 0) otherwise.

4.14 Fork (Library Routine)

```
int fork(fname,arg,mode) char *fname;
```

The resource manager starts a new process running the program found in the file named "fname", which must be in executable load format. The function named "main" is called with the integer argument "arg". "Mode" is a combination (logical "or") of the following flags, defined in "user.h":

one of these: FOREGROUND, BACKGROUND, or DETACHED

and one of these: SHARE, REUSE, EXCLUSIVE, or VIRGIN

If FOREGROUND is specified, then the new process can be killed by entering a control-C on the console. FOREGROUND is mainly used by the command interpreter. If BACKGROUND is specified, then a "process identifier" is returned that may be used to subsequently "killoff" the child. DETACHED (i.e., neither FOREGROUND nor BACKGROUND) is the default. If SHARE is specified, then the resource manager will be willing to start this new process in the same code space as another process executing the same file, if that process was also spawned in SHARE mode. If REUSE is speci-

fied, the code space of an earlier process can be reused. If EXCLUSIVE is specified, then this process may not be started on a machine which already has a process using the same executable file. VIRGIN means that a new copy must be loaded, and is the default. If the call succeeds, a link of type REQUEST and TELLED-EST is given to the resource manager; the child may obtain this link by invoking "parline". The caller may receive messages from the child over this link, which has code 0 and channel CHAN14.

A returned value of -1 indicates an error. Success is indicated by a return value of 0, except in the case of BACKGROUND mode, when the return value is a "process identifier".

4.15 Fsline (Library Routine)

```
int fsline();
```

This routine returns the number of a REQUEST link to be used for communication with the file manager Process. An error gives a returned value of -1.

4.16 Handler (Service Call)

```
handler(vector,func,chan) (*func)();
```

The address of a device vector in low core is specified by "vector". The interrupt vector is initialized so that when an interrupt occurs, the specified routine "func" is called at interrupt level. If the interrupt level routine performs an "awaken" call, a message will arrive on channel "chan" with urcode 0 and urnote "INTERRUPT" (see "receive").

Returned values: Success returns a value of 0. -1 (*)

means that there have been too many handler calls on that machine (the limit is currently 2). -2 (*) means that the channel is invalid. -3 (*) means that the vector address is unreasonable. -4 (*) means that the vector is already in use.

4.17 Inline (Library Routine)

```
int inline();
```

This routine returns the number of a REQUEST link to be used for subsequent terminal input. The terminal driver only allows one input link to be open at any time. An error returns a value of -1.

4.18 Kill (Service Call)

```
kill(lifeline);
```

The process indicated by "lifeline" (the return value of a successful "startup" call) is terminated as if it had performed "die("killed")". The lifeline is not destroyed.

Returned values: Success returns a value of 0. -1 (*) indicates that the link is invalid or not a "lifeline".

Only the resource manager and terminal driver should use this call.

4.19 Killoff (Library Routine)

```
int killoff(procid);
```

This routine asks the resource manager to kill a process that the calling process previously created as a BACKGROUND process with a "fork" request. The value returned from that "fork"

is "procid". The effect on the dead process is as if it had called "die".

0 is returned for success, -1 for failure.

4.20 Link (Service Call)

int link(code,chan,restr)

A new link is created. The caller becomes the new link's owner (forever) and holder (usually not for very long). The caller specifies an integer, "code", which is later useful to the caller to associate incoming messages with that link. The caller also specifies "chan" as one of sixteen possibilities, CHAN1, ..., CHAN16, which are integers containing exactly one non-zero bit. Channels are used to receive messages selectively. CHAN16 should be avoided, for reasons explained in "call". CHAN15 should also be avoided, since the kernel uses it for remote loading. The returned value is the link number that the caller should use to refer to the link. The argument "restr" is the sum of various restriction bits that tell what kind of link it is. The possibilities are:

GIVEALL
DUPALL
TELLGIVE
TELLDUP
TELLDEST
REQUEST
REPLY
MAYERROR

"GIVEALL" means that any holder may give the link to someone else. "DUPALL" means that any holder may duplicate it (i.e., give it to someone with "dup" = DUP; see "send"). "TELLGIVE",

"TELLDUP", and/or "TELLDEST" cause the owner to be notified whenever a holder gives away, duplicates, and/or destroys the link, respectively (see "receive"). A process may duplicate, give away, or destroy a newly created link without restriction and without generating notifications; restrictions and notifications only apply to links received in messages. A link must be either of type "REQUEST" or "REPLY". A REPLY link cannot be duplicated and disappears after one use; a REQUEST link can be used repeatedly unless it is destroyed by its holder. An enclosed link must always be of the opposite type from the link over which it is being sent. If "MAYERROR" is specified, then error messages may be sent along this link. (See "send" and "receive".)

Returned values: The normal return value is a non-negative link number. -1 (*) means that the link was specified as either both or neither of REPLY and REQUEST; -2 (*) means that the channel is invalid, -3 (*) means there is no room for a new link (currently 20 links are allowed to each process).

4.21 Linkok (Service Call)

int linkok(link)

The returned value is 0 if the link number is currently valid, -1 if it is out of range, and -2 if it is in range but does not denote a valid link.

4.22 Load (Service Call)

```
int load(prog,fd,plink,arg) char *prog;
```

This call loads a program. If "fd" is -1, the console operator is requested to load "prog" manually. If "fd" is a valid link number (it should be a link to an open file) and "prog" is -1, the file is loaded on the same machine. In either of these cases, the return value is an "image", to be used for subsequent "startup" or "remove" calls.

If "fd" is a link and "prog" is a machine number, the file is loaded remotely on the corresponding machine and started. The arguments "plink" and "arg" have the same meaning as in the "startup" call. The "plink" is automatically given (not duplicated). The return value is a "lifeline", as for a "startup" call.

Returned values: A nonnegative image number or lifeline number is returned on success. -2 (*) and -3 (*) mean that the link "fd" was out of range or was invalid, respectively. -5 means that there wasn't room for the new image. -6 means that there are too many images. -10 (*) means that the caller had no room for the lifeline. -11 (*) means that the "plink" was out of range or had an invalid destination.

Only the resource manager should use this call.

4.23 Ltodate (Library Routine)

ltodate(n,s) long n; char s[30];

This library routine converts a long integer, representing the number of seconds since Jan 1, 1973, into a readable character string telling the time, day of the week, and date. Dates later than 1999 are not converted correctly.

4.24 Nice (Service Call)

nice()

This call allows the Arachne scheduler to run any other runnable process. (Arachne has a round-robin non-pre-emptive scheduling discipline; "nice" puts the currently running process at the bottom.) It is used to avoid busy waits.

4.25 Open (Library Routine)

int open(fslink,fname,mode) char *fname;

The file named "fname" is opened for reading if "mode" is 0, for writing if "mode" is 1, and for both if "mode" is 2. The argument "fslink" is the caller's link to the file manager. The returned value is a link number, used for subsequent "read", "write", and "close" operations. This link may be given to other processes, but not duplicated. -1 is returned on error.

4.26 Outline (Library Routine)

```
int outline();
```

This routine returns the number of a link to be used for subsequent terminal output. An error returns a value of -1.

4.27 Parline (Library Routine)

```
parline();
```

This routine asks the resource manager for a link to the parent of the caller. It assumes that the parent gave the resource manager a REQUEST link when it spawned the child. An error returns a value of -1.

This call is typically used by a program being run by the command interpreter; the parent link (to the command interpreter) is used to get the command line arguments.

4.28 Print (Library Routine)

```
int print(file,format,args...) char *format;
```

This routine implements a simplified version of UNIX's "printf". The argument "file" is either a link to an open file or a terminal output link. The input is formatted and then "write" is called. The "format" is a character string to be written, except that two-byte sequences beginning with "%" are treated specially. "%d", "%o", "%c", "%w", and "%s" stand for decimal, octal, character, long integer, and string format, respectively. As these codes are encountered in the format, successive "args" are written in the indicated manner. (Unlike

"printf", there are no field widths.) A "%" followed by any character other than the above possibilities disappears, so "%%" is written out as "%". Only 6 arguments are allowed.

4.29 Read (Library Routine)

```
int read(file,buf,size) char *buf;
```

The argument "file" is either a link to an open file or a terminal input link. At most "size" bytes are read into the buffer "buf"; fewer are read if end-of-file occurs. For the terminal, control-D is interpreted as end-of-file. The returned value is the number of bytes actually read.

4.30 Readline (Library Routine)

```
int readline(file,buf,size) char *buf;
```

This routine is the same as "read", except that it also stops at the end of a line. For a file a "newline" character is interpreted as end-of-line; however, "readline" is very inefficient for files. For the terminal, a "line-feed" or "carriage return" terminates a line; the last character placed in the buffer will be "newline" (octal 12). Control-D or control-W will also terminate a line, but they will not be included in the bytes read. The returned value is the number of bytes read.

4.31 Recall (Library Routine)

```
int recall(inmess,inlen) char *inmess; int *inlen;
```

If a previous "call" (or "recall") returned a value of -3, meaning that the message did not arrive in 5 seconds, a process can invoke the library routine "recall" to continue waiting. Only the return message buffer and place to store the length are specified (cf. "call").

Returned values: These are the same as for "call", except that -2 and -6 don't apply.

4.32 Receive (Service Call)

```
int receive(chans,data,urmess,delay) char *data;

struct urmesg {
    int urcode; /* for receiving messages */
    int urnote; /* chosen by user, see "link" */
    int urchan; /* filled in by Arachne, see "receive" */
    int urlnenc; /* chosen by user, see "link" */
    int urlnenc; /* index of enclosed link */
    int urlength; /* length of incoming message */
} *urmess;
```

The caller waits until a message arrives on one of several channels, the sum of which is specified by "chans". All other messages remain queued for later receipt. The code and channel of the link for the incoming message are returned in "urcode" and "urchan", respectively.

The value of "urnote" is one of six possibilities: DUPPED, DESTROYED, GIVEN, INTERRUPT, DATA, or ERROR. The first three of these mean that the link's holder has either duplicated, destroyed, or given away the link (see "send" and "link"). In the case of "DESTROYED", the body of the message may contain data

placed there during termination of the sender (see "die" and "kill"). "INTERRUPT" is discussed under "handler". "DATA" means that the message was sent by "send".

"ERROR" means either that the message was sent by "send", but the link had "MAYERROR" and the sender specified "ERROR", or the link had "MAYERROR" but not "TELLDEST" and the holder terminated (see "link"). Receipt of an error message raises an exception (see "errhandler").

The newly assigned link number for the link enclosed with the message is reported in "urlnenc"; the caller now holds this link). If no link was enclosed, "urlnenc" is -1. The length of the incoming message is reported in "urlength". The argument "data" must point to a buffer of size MSLEN into which the incoming message, if any, will be put. The caller may discard the message by setting "data" to zero. The argument "delay" gives the time in seconds that the caller is willing to wait for a message on the given channels; a "delay" of 0 means that the call will return immediately if no message is already there, and a "delay" of -1 means that there is no limit on how long the caller will wait. A process can sleep for a certain amount of time by waiting for a message that it knows won't come (e.g., on an unused channel).

Returned values: 0 is returned on success. -1 (*) means the caller has no room for the enclosed link (see link; the message can be successfully received later), -2 (*) means that the argument "urmess" was bad, -3 means that the waiting time expired.

4.33 Remove (Service Call)

`remove(image)`

The code segment indicated by "image", the return value of a successful "load" call, is removed. Only the process that performed a "load" is allowed to subsequently "remove" that image.

Returned values: Success returns a value of 0. -1 (*) means that the image either doesn't exist or is in use, or that the caller didn't originally load the image.

The resource manager uses this call to create space for new images; no other program should use this call.

4.34 Seek (Library Routine)

`int seek(file,offset,mode)`

The argument "file" is a link to an open file. The current position in the file is changed as specified by the "offset" and "mode". A value for "mode" of 0, 1, or 2 refers to the beginning, the current position, or the end of the file, respectively. The "offset" is measured from the position indicated by "mode"; it is unsigned if "mode" = 0, otherwise signed. A returned value of 0 indicates success, -1 indicates failure.

4.35 Send (Service call)

`int send(ulink,elink,data,length,dup) char *data;`

This call sends a message along link number "ulink". The message body is "data" and its length is "length". If no message is to be sent, either "data" or "length" should be zero. If the

caller wishes to pass another link that it holds with the message, it specifies that link's number in "elink" (the "enclosed link"). If there is no enclosure, "elink" should be -1. The use of elinks is restricted in various ways; see "link".

The argument "dup" specifies either "DUP" or "NODUP"; in the first case, the enclosed link is duplicated so that both the sender and receiver will hold links to the same owner; in the second case, the enclosed link is given away so that only the receiver of the message will hold it.

The "dup" argument also may specify "ERROR" (this bit should be or'ed into "DUP" or "NODUP"). If "ulink" has the "MAYERROR" restriction, then an "ERROR" message will be sent to the recipient. If "MAYERROR" is not set, then "ERROR" has no effect.

Returned values: 0 is returned on success. -1 (*) means that the ulink number is bad and -2 (*) means that the ulink is invalid. -3 (*) and -4 (*) have corresponding meanings for the elink. -5 (*) means that the message was bad, -6 (*) means that the elink can't be duplicated, -7 (*) means that the elink can't be given away, and -8 (*) means the message is too long.

No error is reported if the destination process has terminated; in this case, the message is discarded.

4.36 Setdate (Service Call)

setdate(n) long n;

This service call sets the wall clock to "n", which is a long integer representing the number of seconds since midnight,

Jan 1, 1973.

Only the command interpreter and resource manager should use this call.

4.37 Startup (Service Call)

```
int startup(image,arg,plink,dup,fd)
```

This call starts a process whose code segment is indicated by "image", the return value of a successful "load" call. The child is given "arg" as its argument to "main". The child's link number 0 is "plink", a link owned by the caller; this link is either given to the child or duplicated depending on whether "dup" is NODUP or DUP, respectively. The child cannot destroy link 0. For C programs, the data area is part of the image; for Elmer programs, "startup" causes a new data area to be created. The "fd" argument should be a link number for an open file that holds the Elmer program; it is used to load the data segment.

Returned values: Success returns a non-negative lifeline number, which can be used for a subsequent "kill". -1 (*) means that the caller had no room for the lifeline (see "link"). -2 (*) or -3 (*) means that the "plink" was out of range or had an invalid destination, respectively. -4 means that there was no room for the new process' stack (or data area: Elmer only). -5 (*) means that the "image" was invalid, -6 (*) means that "image" is an Elmer program, and "fd" is bad.

Only the resource manager should use this call.

4.38 Stat (Library Routine)

```
int stat(fmlink,fname,statbuf) char statbuf[36];
```

This library routine gives information about the file named "fname". The argument "fmlink" is the caller's link to the file manager. An error returns a value of -1. After a successful call, the contents of the 36-byte buffer "statbuf" have the following meaning:

```
struct{
    char    minor;          minor device of i-node
    char    major;          major device
    int     inumber;
    int     flags;
    char    nlinks;          number of links to file
    char    uid;             user ID of owner
    char    gid;             group ID of owner
    char    size0;           high byte of 24-bit size
    int     size1;           low word of 24-bit size
    int     addr[8];         block numbers or device number
    long    actime;          time of last access
    long    modtime;         time of last modification
} *buf;
```

NOTE:

Some of these fields are irrelevant, since all Arachne files have the same owner.

4.39 Time (Service Call)

```
long time();
```

This service call returns a long integer that may be used for timing studies. The integer is a measure of time in intervals of ten-thousandths of seconds. NOTE: The time wraps around after a full double word (32 bits).

4.40 Unlink (Library Routine)

int unlink(fslink, fname) char *fname;

This library routine removes the file named "fname"; it cleans up after "create" and "alias". The argument "fslink" is the caller's link to the file manager. Error returns a value of -1.

4.41 Write (Library Routine)

write(file, buf, size) char *buf;

The argument "file" is either a link to an open file or a terminal output link. Using this link, "size" bytes are written from the buffer "buf". There are no return values.

5. CONSOLE COMMANDS

The Command Interpreter is a utility process that reads the teletype. When the Command Interpreter is awaiting a command, it types the prompt ".". A command consists of a sequence of "arguments" separated by spaces. Otherwise, spaces and tabs are ignored except when included in quotation marks ("). Within quotes, two consecutive quotes denote one quote; otherwise, quotation marks are deleted. The first "argument" is interpreted as a "command" (see below). Command names may be truncated, provided the result is unambiguous. It is intended that all commands will differ in their first three characters.

The "run" command may be followed by from one to MAXCOMS (4)

commands separated by the symbol "^". The terminal output of the command to the left of a "^" is buffered by a special "pipe" process, and fed as though it were terminal input to the command to the right of the "^". The output from the last process in a pipe may be redirected to a file by following it with " ^ to outfilename". The input to the first process in a pipe may be obtained from a file by preceding it with "from infilename ^ ". Although "to" and "from" appear to be the names of processes in the pipe, they do not count towards the MAXCOMS maximum. Furthermore, "to" and "from" are reserved words to the command interpreter, and hence neither may be the name of a user program.

The following is an alphabetized list of console commands.

5.1 alias <filename1> <filename2>

The second indicated file becomes another name for the first indicated file. If either of these is "deleted", the other (logical) copy still exists; however, changes to either affect both.

5.2 background <filename> <arg>

The indicated file must be executable. It is started as a BACKGROUND process, with the integer argument "arg". The Command Interpreter prints out the new process's process identifier, which may be used for subsequent "killing" and then gives the next prompt.

5.3 copy <filename1> <filename2>

The second indicated file is created with a copy of the contents of the first indicated file.

5.4 delete <filename>

The indicated file is deleted.

5.5 dump <address>

Prints a screenful of memory locations in octal for debugging.

5.6 help

A list of available commands is displayed.

5.7 kill <arg>

The indicated argument should be the process identifier returned from a previous "background" command. The process referred to by the process identifier is killed.

5.8 make <filename>

The named file is created. Subsequent input is inserted into the file; the input is terminated by a control-D.

5.9 rename <oldname> <newname>

The name of file "oldname" is changed to "newname".

5.10 run <filename> { <arg> } { ^ <filename> { <arg> } }

The indicated files should be executable files. The right-most one is run as a FOREGROUND process. The others are run as BACKGROUND processes. The Resource Manager is given a REQUEST link, which the new process may use to ask for the command line arguments. When the loaded program starts up, the argument to "main" tells the number of command line arguments. To get the individual arguments, the loaded program sends a message to the Command Interpreter (its parent). The first word of the message is ARGREQ, and the second is an integer specifying which argument is desired. The name of the program is argument number 0. The returned message body is the argument, which is a null-terminated string of length at most MSLEN.

5.11 set <modelist> or SET <modelist>

This command changes the console input modes. The mode list is a sequence of keywords "x" or "-x", where "x" can be any of the following:

upper	(the terminal is upper case)
echo	(the terminal echoes input)
hard	(the terminal is hard-copy)
tabs	(the terminal has hardware tabs)

Keywords may be abbreviated according to the same rules as commands. The format "x" turns on the corresponding mode, "-x" turns it off. (UPPER is recognized for upper; "lower" means "upper".) For more information, see the section "CONSOLE INPUT PROTOCOLS".

5.12 time <format>

If a format is given (as "yymmddhhmm"), the wall clock is set to that time and printed. With no argument, "time" prints the wall clock time.

5.13 type <filename>

The indicated file is typed.

6. TERMINAL INPUT PROTOCOLS

The terminal driver performs interrupt-driven I/O, which allows for typing ahead. Also, the following characters have special meanings:

Control-C	kill the running program (but don't kill the command interpreter itself)
Control-D	end of file (terminates a "read" or "read-line")
Control-W	end of line (but no character sent)
line-feed	end of line
carriage return	end of line
rubout	erase last character (unless line empty)
Control-X	erase current line
Control-S	enter scroll mode; pause every 18 lines of

output; if paused, allow the next 18 lines to be printed.

Control-Q leave scroll mode; if paused, allow output to continue.

escape next character should be sent as is

In "echo" mode, input is echoed, otherwise not. In "hard" mode, output is designed to be legible on hardcopy devices; otherwise the terminal driver assumes that the cursor can move backward, as on a CRT. In "tabs" mode, advantage is taken of hardware tabs on the terminal. In "upper" mode, the terminal is assumed to only have upper case. Input is converted to lower case, unless escaped. Upper case characters are printed and echoed with a preceding "!". Escaped [,], @, ^, and \ are converted to {, }, `, ~, and |, respectively, and the latter are similarly indicated by preceding "!"s.

7. UTILITY PROCESS PROTOCOLS

This section describes the protocols that user programs must follow to communicate with the utility processes when the library routines described earlier are inadequate. Four utility processes are the resource manager, the file manager, the terminal driver, and the command interpreter. The resource manager keeps track of which programs are loaded and/or running on the local machine. The kernel and the resource manager reside on each machine. The terminal driver governs I/O on the console; the command interpreter interprets console input. The file manager implements a file system by communicating with the PDP-

11/40. It need not exist on every machine.

During Arachne initialization, one resource manager is started. It loads a full complement of utility processes (the terminal driver, command interpreter, and file manager) on its machine and various utility processes on the other machines. When a particular resource manager is not given a local terminal driver or file manager, it shares the one on the initial machine.

7.1 Input/Output Protocols

This section describes the message formats used for communicating with the file manager and terminal driver processes. A program that explicitly communicates with the file manager or terminal driver must include the header files "filesys.h" and "ttddriver.h", which define the necessary structures.

To open an input or output line to the terminal, to change the modes on the terminal, or to inform the teletype of whom it should kill when encountering a control-C, a message is sent over the terminal link of the following form:

```
struct ttinline{
    char tticom;
    char ttisubcom;
    char ttimodes;
}
```

"tticom" is either OPEN, STTY, MODES, or TOKILL. In the case of OPEN, "ttisubcom" is either READ or WRITE, and the return message has the new link enclosed. In the case of STTY, "ttimodes" tells what the new modes should be (a bit-wise sum of ECHO, TABS, HARD, and UPPER). In the case of MODES (to find out the current modes), the return message has the modes in "ttimodes". In the

case of TOKILL (to inform the terminal driver which process to kill on receipt of control-C), the message encloses a lifeline.

To open, create, unlink, alias, or get status information on a file, a message is sent over the file manager link in the following form:

```
struct ocmsg{
    int ocaction;
    int ocmode;
    char fsname[MSLEN-4];
}
```

"ocaction" is either OPEN, CREATE, UNLINK, ALIAS, or STAT. "ocmode" is the mode for OPEN or CREATE; in the case of ALIAS, it holds the length of the first file name. "ocname" contains the file name (or, in the case of ALIAS, the concatenation of two file names), null-terminated. In the cases of OPEN or CREATE, a successful return contains a valid enclosed link; for UNLINK, STAT, or ALIAS, there is no enclosed link. In the case of STAT, the return message has the structure of a "rdmsg" as in the case of READ below; the response has length 36 or 0, corresponding to success or failure, respectively. In all other cases, the response is one word: 0 on success, -1 on failure.

For either the terminal or the file manager, reading or writing is done by sending a message of the following form:

```
struct fsmesg{
    int fsaction;
    int fslength;
}
```

"fsaction" should be either READ, READLINE, or WRITE. "fslength" tells how many bytes are intended to be read, or are being sent to be written. In the case of WRITE, the text is sent in subse-

quent messages, and nothing is returned. In the cases of READ or READLINEy s the response is of the following form:

```
struct rdmesg{
    char rdtext[MSLEN];
}
```

The maximum allowable read is size MSLEN. The actual size of the returned message is contained in the "urlength" field.

To perform a seek on an open file, send a message to the file manager of the following form:

```
struct skmesg{
    int skaction; /* should be SEEK */
    int skoffset;
    int skmode;
}
```

The return message is one word: 0 for success, -1 for failure.

7.2 Resource Manager Protocols

Processes that communicate explicitly with the resource manager must include the header file "resource.h". The following structure is declared there:

```
struct rmmesg { /* messages to resource managers */
    int rmreq; /* type of request */
    int rmarg; /* various miscellaneous arguments */
    int rmmode; /* the mode for STARTs or KILLs */
}
```

The resource manager keeps track of which images (code segments) and processes exist. A separate resource manager runs on each machine in the network; these programs communicate with each other, but are relatively independent.

Each resource manager holds a terminal link and file manager link, which are either for local utility processes or else links received from the first resource manager initialized. Whenever a

resource manager has a local terminal it also has a local command interpreter.

There are three kinds of processes: FOREGROUND, BACKGROUND, and DETACHED. When a process is started, its link 0 is owned by the local resource manager, to whom all of this process's requests are directed.

The first FOREGROUND process for any terminal is always the command interpreter, which initially "has the ball". Each terminal always has one FOREGROUND process that "has the ball". The process "with the ball" may create another FOREGROUND process, which means that the child now "has the ball". The meaning of "having the ball" is that a control-C entered on the corresponding terminal will terminate the process. When the process "with the ball" terminates, its parent then "recovers the ball", and will be terminated by the next control-C. If one of the processes in this FOREGROUND chain terminates, the chain is re-linked appropriately. The command interpreter is an exception in that control-C's have no effect on it.

A process may also create another process as a BACKGROUND process. In this case, the child's process identifier is returned to the parent, and later the parent can use this identifier to terminate the child. These identifiers are assigned by the resource manager, and are distinct from the process identifiers used in the kernel.

A DETACHED process cannot be terminated by either method.

A user may make five kinds of requests on its resource manager:

1. RMTTREQ Request

The resource manager is requested to give the requestor a link to the requestor's terminal. This link will be sent over the enclosed link in the request, which should therefore be a REPLY link. .

2. RMFSREQ Request

The resource manager duplicates its file manager link and sends it back over the enclosed link in the request, which should therefore be a REPLY link.

3. RMSTART Request

The resource manager will start a process, using the link enclosed with this request for two purposes: 1) to respond to the request (see conditions for response below), or 2) to save it and give to the child if the child asks for it (see RMPLINK below). The caller must be careful, of course, not to give a REPLY link if both uses are intended. Also, the caller must make the enclosed link GIVEALL if the resource manager should try to load the process on another machine, rather than giving up if it doesn't fit on the local one. The RMSTART request also specifies the file name and an integer argument to be given to the child when it starts.

The caller also specifies a "mode" for starting the child, which is a combination of bits with various meanings. The user should specify either BACKGROUND, FOREGROUND, or DETACHED (the

default is DETACHED). FOREGROUND is only allowed if the requester currently "has the ball" for its terminal. The user may specify EXCLUSIVE, which causes the resource manager to load it on a machine only if there is no like-named core image with its EXCLUSIVE bit set, on that machine. The user should specify either SHARE, REUSE, EXCLUSIVE, or VIRGIN (the default is VIRGIN). These alternatives are described above (see "fork"). The user should also specify either GENTLY or ROUGHLY (the default is GENTLY). If GENTLY, the resource manager will first try to load it locally without throwing out any other unused images and then will try to do the same on other machines. When this fails, or if ROUGHLY was specified, it tries to make room locally for the new process, and then tries to do so on other machines. The user should also specify either ANSWER or NOANSWER (the default is NOANSWER). If ANSWER is specified, or if BACKGROUND was specified, then the resource manager sends a reply over the enclosed link. The first word of the reply is the return code; -1 always means failure; 0 means success except in the case of BACKGROUND, when the value returned is the process identifier of the child.

An existing code segment is reusable if the filename still refers to an existing publicly executable load format file that has not been modified since the copy in question was loaded. Any number of processes may share a code segment. The terminal associated with a child process is always the same as the one associated with its parent; the command interpreter is loaded with a terminal during initialization.

4. RMKILL Request

The resource manager kills the process whose process identifier is given as part of the request. The request may enclose a link that is used to give a one-word acknowledgement of success or failure if the request specifies ANSWER (as in RMSTART, described above). The process being killed must of course be BACKGROUND, and only the process that started it is allowed to kill it.

5. RMPLINK Request

The resource manager returns the link that was originally enclosed with the request that started this process. It is returned over the link enclosed with the RMPLINK request, which must therefore be of the proper type, whichever that may be.

8. ACKNOWLEDGEMENTS

The authors would like to acknowledge the assistance of Professor Sun Zhong Xiu of Nanking University, Peoples Republic of China, and the following graduate students who have been involved in the Arachne project: Jonathan Dreyer, Jack Fishburn, Michael Horowitz, Will Leland, and Paul Pierce. Their hard work has helped Arachne to reach its current level of development.

9. REFERENCES

- Baskett, F., Howard, J. H., Montague J. T., "Task Communication in Demos", Proceedings of the Sixth Symposium on Operating Systems Principles, pp. 23-31, November 1977.
- Finkel, R. A., Solomon, M. H., The Roscoe Kernel, Version 1.0, University of Wisconsin--Madison Computer Sciences Technical Report #337, September 1978.
- Finkel, R. A., Solomon, M. H., Tischler, R. L., Roscoe Utility Processes, University of Wisconsin--Madison Computer Sciences Technical Report #338, February 1979.
- Finkel, R. A., Solomon, M., and Tischler, R., "The Roscoe Resource Manager", Proceedings of Compcon Spring 1979, pp. 88-91, February, 1979.
- Finkel, R. A., Solomon, M. H., Tischler, R., Roscoe User Guide, Version 1.1, University of Wisconsin--Madison Mathematics Research Center Technical Report #1930, March 1979.
- Finkel, R. A., Solomon, M. H., Tischler, R., Arachne User Guide, Version 1.2, University of Wisconsin--Madison Computer Sciences Technical Report #379, February 1980.
- Finkel, R. A., Solomon, M. H., The Arachne Kernel, Version 1.2, University of Wisconsin--Madison Computer Sciences Technical Report #380, February 1980.
- Kernighan, B. W., Ritchie, D. M., The C Programming Language, Prentice-Hall, 1978.
- Ritchie, D. M., Thompson, K., "The UNIX Time-Sharing System", Communications of the ACM, Vol. 17, No 7, pp. 365-375, July 1974.
- Solomon, M. H., Finkel, R. A., ROSCOE -- a multiminicomputer operating system, University of Wisconsin--Madison Computer Sciences Technical Report #321, September 1978.
- Solomon, M., and Finkel, R., "The Roscoe Distributed Operating System", Proceedings of the Seventh Symposium on Operating Systems Principles, pp. 108-114, 10-12 December, 1979.
- Tischler, R. L., Finkel, R. A., Solomon, M. H., Roscoe User Guide, Version 1.0, University of Wisconsin--Madison Computer Sciences Technical Report #336, September 1978.

END

FILMED

6-83

DTIC