AD-A179 384 PERFORMANCE EVALUATION OF PARALLEL BRANCH AND BOUND SEARCH WITH THE INTEL..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. R T MRAZ 1/3

UNCLASSIFIED DEC 86 AFIT/GCE/ENG/86D-2 F/G 9/2 NL

DTIC FILE COPY

AD-A179 384

DTIC
ELECTE
APR 1 7 1987
S D
D

PERFORMANCE EVALUATION OF PARALLEL BRANCH AND BOUND SEARCH WITH THE INTEL iPSC HYPERCUBE COMPUTER

THESIS

Richard T. Mraz
Captain, USAF

AFIT/GCE/ENG/86D-2

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

87 4 16 042

AFIT/GCE/ENG/86D-2

DTIC
ELECTE
APR 17 1987
S D
D

PERFORMANCE EVALUATION OF PARALLEL BRANCH AND BOUND SEARCH WITH THE INTEL iPSC HYPERCUBE COMPUTER

THESIS

Richard T. Mraz
Captain, USAF

AFIT/GCE/ENG/86D-2

Approved for public release; distribution unlimited

AFIT/GCE/ENG/86D-2

PERFORMANCE EVALUATION OF PARALLEL BRANCH AND BOUND SEARCH WITH THE INTEL iPSC HYPERCUBE COMPUTER

THESIS

Presented to the Faculty of the School of Engineering

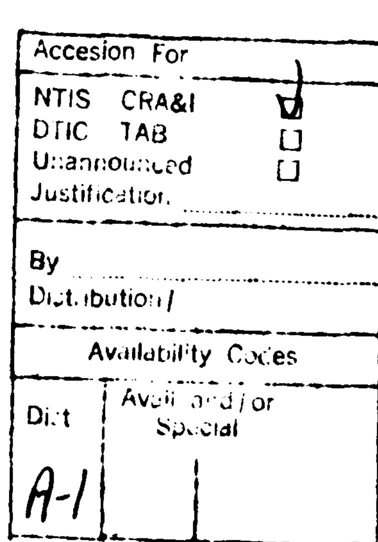
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Engineering in Computer Engineering


Accesion For
NTIS CRA&I
DTIC TAB
Unannounced
Justification
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

Richard T. Mraz, B.S.

Captain, USAF

December 1986

Approved for public release; distribution unlimited

Contents

		Page
List of Figures		v
List of Tables		vii
Abstract		viii
I.	Introduction	1
	Background	2
	Problem	4
	Classes of Search	6
	Parallel Processing Issues	8
	Maximum Parallelism	8
	Problem Limitations	9
	Algorithm Limitations	9
	Architecture Limitations	10
	Parallel Design	10
	Overview of the Thesis	11
II.	Intel iPSC Hypercube	12
	iPSC Design Philosophy	12
	Hypercube Interconnection	13
	Intel iPSC	15
	iPSC Software Development	16
	Conclusions	17
III.	Analysis of Parallel Design	18
	Parallel Abstract Data Types	18
	Abstract Data Type	19
	Parallel ADT	20
	Analysis of the PADT	20
	Design Methodologies	21
	Traditional Design	22
	Top-Down Structured Design	22
	Data-Structure Design	23
	Object-Oriented Design	23
	Analysis of Design Methodologies	23
	Parallel Models of Computation	24
	Data-Flow Model of Computation	25
	Control-Flow Model of Computation	26
	Analysis of the Data-Flow and Control-Flow Models	27
	Process Model of Computation	27
	Object-Oriented Design	28
	Class	28
	Message	29
	Method	30
	Object	30

		Page
	Object Design Approach	31
	Define the Problem	31
	Identify Objects and Attributes	31
	Identify Operations	32
	Establish Visibility	32
	Establish Interface	32
	Implement the Objects	32
	Conclusions	34
IV.	Parallel Branch & Bound	35
	Define the Problem	35
	Branch & Bound Search	38
	Identify Objects and Attributes	42
	Identify Operations	43
	Establish Visibility	44
	Establish Interface	50
	Implement Objects	53
	Conclusions	53
V.	Parallel N-queens and Parallel Deadline Job Scheduling Implementation	54
	Parallel N-queens Constraints	55
	N-queens Control Process	58
	Overview	58
	Meta-Controller	60
	Terminate Check	60
	N-queens Worker Process	61
	Overview	61
	Controller	61
	Problem Solver	61
	E-Node Expander	61
	Bound Check	62
	Terminate Check	62
	Parallel Deadline Job Scheduling Constraints	63
	Deadline Job Scheduling Control Process	68
	Overview	68
	Meta-Controller	68
	Terminate Check	70
	Bound Check	70
	Deadline Job Scheduling Worker Process	71
	Overview	71
	Controller	71
	Problem Solver	71
	E-Node Expander	72
	Bound Check	72
	Terminate Check	72
	Conclusions	73

		Page
VI.	Performance Analysis and Experiment Results	74
	Computation Time	74
	Speed Up	75
	Load Balance	76
	Baseline Performance	76
	Parallel Performance Experiments	77
	Parallel N-queens	78
	Parallel Deadline Job Scheduling	87
	Conclusions	96
VII.	Conclusions and Recommendations	97
	Parallel Design Methodology	97
	Performance of Parallel Branch and Bound	98
	Suitability of Hypercube Architectures for Parallel Search	99
	Recommendations	99
Appendix A: iPSC N-queens Source Code Description		101
Appendix B: Sequential N-queens Source Code Description		123
Appendix C: iPSC Deadline Job Scheduling Source Code Description		131
Appendix D: Sequential Deadline Job Scheduling Source Code Description		160
Appendix E: Tables of Branch and Bound Experiments		176
Bibliography		187
Vita		190

List of Figures

Figure Page

1. U.S. Companies Offering or Building Parallel Processors 3
2. Example of Processors Communicating using Message Passing 13
3. Node Numbers for a 3-dimension cube 14
4. Three-Dimension cube structure with vertices labeled from 0 to 7 in binary 14
5. STACK Abstract Data Type 19
6. Functionality of a Design Methodology 22
7. Model of Computation 24
8. Execution of a data-flow computation (a+b)/(c+d) 26
9. Execution of a control-flow computation (A+B)/(C+D) 27
10. Example Class Hierarchy 29
11. High Level Language/Object Constructs Matrix 33
12. Search Strategies 36
13. Search Tree with Node Definitions 37
14. Example Search Tree 37
15. Branch & Bound Data Flow Diagram 41
16. Sequential Branch and Bound Visibility Diagram 45
17. Parallel Visibility Diagram #1 46
18. Parallel Visibility Diagram #2 47
19. Parallel Branch & Bound Visibility Diagram used for this research 49
20. Board Positions for the solution vector (1,4,2,3) 55
21. Partial Solution Space for the 4-queens problem 56
22. Board Positions for the answer vector (2,4,1,3) 57
23. Object Diagram for Parallel Branch & Bound 59
24. Example 4-Job Deadline Job Scheduling Solution Space 65

Figure Page

25. Object Diagram for Parallel Branch & Bound 69

26. N-queens Time to First Solution 80

27. N-queens Time to All Solutions 81

28. iPSC and Elxsi Speed Up Over VAX 83

29. Time to All Solutions Optimized for Board Sizes of 8, 9, & 10 84

30. Load Balance for the 11-queens on a D-5 Hypercube 86

31. Deadline Job Scheduling- Problem Set #1 Computation Time 90

32. Deadline Job Scheudling- Problem Set #1 Load Balance of scheduling 20-Jobs on an iPSC D-4 91

33. Deadline Job Scheduling- Problem Set #2 Computation Time 93

34. Deadline Job Scheduling- Problem Set #2 Speed Up Over VAX 94

35. Deadline Job Scheudling- Problem Set #2 Load Balance of scheduling 15-Jobs on an iPSC D-4 95

36. Portion of the 4-queens solution space generated during search 129

37. Example 4-Job Deadline Job Scheduling Solution Space 171

38. Portion of the 4-Job solution space generated during search 173

List of Tables

Table — Page

1. N-queens VAX Baseline 176
2. N-queens Elxsi Computation Time 176
3. N-queens iPSC d-5 Computation Time 177
4. N-queens iPSC d-4 Computation Time 177
5. N-queens iPSC d-3 Computation Time 178
6. N-queens iPSC d-2 Computation Time 178
7. N-queens iPSC d-1 Computation Time 178
8. N-queens iPSC d-0 Computation Time 179
9. Load Balance for the 11-queens Problem on a d-5 cube 179
10. Deadline Job Scheduling VAX Baseline 180
11. Deadline Job Scheduling iPSC d-5 Computation Time 181
12. Deadline Job Scheduling iPSC d-4 Computation Time 182
13. Deadline Job Scheduling iPSC d-3 Computation Time 183
14. Deadline Job Scheduling iPSC d-2 Computation Time 184
15. Deadline Job Scheduling iPSC d-1 Computation Time 185
16. Load Balance - Deadline Job Scheduling Problem Set #1 20-Jobs Solved on an iPSC d-4 186
17. Load Balance - Deadline Job Scheduling Problem Set #2 15-Jobs Solved on an iPSC d-4 186

Abstract

With the recent availability of commercial parallel computers, researchers are examining new classes of problems for benefits from parallel processing. This report presents results of an investigation of the set of problems classified as search intensive. The specific problems discussed in this report are the 'backtracking' search method of the N-queens problem and the Least-Cost Branch and Bound search of deadline job scheduling. The object-oriented design methodology was used to map the problem into a parallel solution. While the initial design was good for a prototype, the best performance resulted from fine tuning the algorithms for a specific computer. The experiments of the N-queens and deadline job scheduling included an analysis of the computation time to first solution, the computation time to all solutions, the speed up over a VAX 11/785, and the load balance of the problem when using an Intel Personal SuperComputer (iPSC). The iPSC is a loosely couple multiprocessor system based on a hypercube architecture. Results are presented which compare the performance of the iPSC and VAX 11/785 for these classes of problems.

Performance Evaluation of Parallel Branch and Bound Search on the Intel iPSC Hypercube Computer

I. Introduction

Within the past decade, parallel computer architectures have been a subject of significant research effort. Integrated circuit technology, high speed communications, along with hardware and software technology have made parallel computers much easier to build and much more reliable (9,22,26,28). Parallel processing has also proven to be an effective solution to certain classes of problems. Probably the most notable class is array or vector problems that run order-of-magnitudes faster on parallel architectures such as the Cray. Because of the recent proliferation of parallel computers, researchers are investigating other classes of problems for potential benefits from parallel architectures. Problems classified as search intensive is one such class. Two organizations sponsoring research in parallel computing with problems that are typically search intensive are the Strategic Defense Initiative Organization (SDIO) and the Defense Advanced Research Projects Agency (DARPA) (6,7,24,29,32).

The SDIO is investigating defensive weapon systems and battle management systems for a strategic defense (6, 29). While the 'hardware' of the strategic defense initiative such as kinetic energy weapons, laser technology, and particle beams seem plausible, the computer technology, algorithms, and distributed control of a strategic defense is far from reality. As Seward and Davis point out, the SDI will "require subsystems whose complexities are several orders of magnitude greater than those that have been developed or proposed in the past" (19:2).

DARPA, on the other hand, initiated a program called the Strategic Computing Program in 1983 (7:100). This program involves research into parallel computers as well as artificial intelligence techniques for military applications. The Air Force component of the Strategic Computing Program is called the Pilot's Associate (PA). Researchers for the Pilot's Associate are investigating flight domain systems that provide expert advice in critical mission functions, such as aircraft systems monitoring, situation assessment, mission planning, and tactics advising (24:102).

Initial goals of the SDI and Pilot's Associate include, quantifying the practical value of and understanding the complexity of such systems as well as defining the specifications for an operational strategic defense or pilot's associate. Once the problems are understood and the accuracy of such systems has been proven, then researchers intend to "speed up" these systems by using supercomputers or parallel computer architectures (32:74, 6:277-278).

Background

According to Händler, the term parallel processing describes the different kinds of simultaneous operations within a digital computer (13:1). While typical VonNeumann architectures emphasize sequential process and control, parallel computers attempt to increase machine performance by exploiting the independence of subtasks within a problem or the independence of the control within a problem. Research in parallel computation traditionally involves many areas, including (31:1102);

- Communications Networks
- Distributed Operating Systems
- Fault Tolerant Hardware and Software
- Distributed Control
- Parallel Algorithms
- Distributed Programming Languages

Given a particular problem and a specific parallel computing environment, these areas may depend on one another. Because of such dependencies, parallel computers do not guarantee increased performance. For example, the distributed operating system relies on the communications network for transmissions of information and control among the processing elements. An inefficient communications subnet increases the time for error free data transmission. This could easily lead to a decrease in system performance. On the other hand, a good parallel algorithm matched with the proper parallel computer architecture could improve performance by an order of magnitude. Kleinrock summarized the paradox by saying, "we have the potential for this power [increased performance] --if only we could figure out how to put all the pieces together!" (18:1200).

Recently, research in the area of parallel computers has been highly successful in several general purpose hardware designs (see Figure 1).

Company	Product
Alliant Computer Systems Corporation	FX/Series
Bolt, Beranek, and Newman	Butterfly
Control Data Corporation	Cyber 205 Series 600
Cray Research Inc.	Cray-2 and X-MP
Digital Equipment Corporation	VAX 11/782 and 784
ELXSI (a subsidiary of Trilogy Inc.)	System 6400
Encore Computer Corporation	Multimax
ETA Systems Inc. (a spin-off of Control Data Corporation)	GF-10
Floating Point Systems Inc.	T Series
Goodyear Aerospace Corporation	MMP
IBM Corporation	Research Parallel Project RP3
Intel Scientific Computers	iPSC
Schlumberger Ltd.	FAIM-1
Sequent Computer Systems Inc.	Balance 21000
Thinking Machines Corporation	Connection Machine

Figure 1: U.S. Companies Offering or Building Parallel Processors (9:753)

Clearly, this list indicates the availability of parallel processing system hardware; however, the application and software support systems are not as prevalent. Stankovic points out that "much of the distributed system software research is experimental work" (30:17). He further emphasizes that "work needs to be done in the evaluation of these systems in terms of the problem domains they are suited for and their performance" (30:17).

Yet, another parallel processing problem is the mapping of a problem to a parallel solution. Probably the largest problem researchers face today in parallel computer systems is the inability of humans to decipher the inherent parallelism of problems that are traditionally solved using sequential algorithms. Patton identified a possible cause of this human shortcoming when he said, "While the world around us works in parallel, our perception of it has been filtered through 300 years of sequential mathematics, 50 years of the theory of algorithms, and 28 years of Fortran programming" (22:34). Basically, humans have not trained their thought processes to accommodate the concepts of solving problems in parallel. Because of this, without new parallel computing algorithms, parallel software development tools, and performance measuring techniques, parallel computing may never be fully exploited.

Problem

Because of the proliferation of parallel computers and because a large class of problems that may benefit from parallel processing are search intensive, this research investigates the actual performance of the class of search problems on a parallel computer.

Two examples of the need for research into parallel search algorithms and performance evaluations are elements of the Strategic Defense Initiative and the Pilot's Associate. Specifically, the basic problems faced by SDI and PA researchers fall into the same class of problems, search intensive processing (see below, Classes of Search). This type of processing

is characterized by large solution spaces that must be examined for answers and exponential time complexity to find a solution. For example, the SDI battle management system must resolve the resource allocation of sensor and tracking satellites to defensive weapon systems (29:4-5). Answers to such a search problem involves a complex solution space with exponential computation time. Researchers plan to reduce the run time complexity using parallel computers. The ultimate goal is to find the proper combination of parallel computer architecture and parallel algorithm such that results can be calculated in 'real-time.' Where 'real- time' is that time interval in which an answer must be delivered (21:8). The general approach to solve some of the battle management and PA problems uses traditional operations research (OR) and artificial intelligence (AI) programming techniques. These techniques are, in general, based upon a systematic search of the solution space of the problem. Hence, this research focuses on parallel search methods. And without losing generality, the specific technique is parallel branch and bound.

The parallel environment for this research is the Intel iPSC Hypercube computer. The iPSC is used for three reasons, (1) the iPSC is available for parallel computing research at the Air Force Institute of Technology; (2) the iPSC is a general purpose parallel architecture, and (3) the iPSC has a flexible software development environment to allow comparisons against sequential implementations. The methodology of programming the iPSC as well as the ability to create several communications subnet configurations make it a flexible parallel architecture (see Chapter II, Intel iPSC Hypercube, for details). The goals of this research can be summarized as follows,

1- Explore a design methodology to map a problem onto a parallel computer.

   Because of the difficulties of mapping a problem to a parallel computer, a formal design approach is needed to help the programmer identify the parallel activity within a problem. Since the development and proof of a new design methodology is beyond the scope of this research, only traditional design approaches will be examined.

2- Measure the performance of parallel branch and bound on parallel computer.

Since some researchers with search intensive problems, such as the SDIO and the Pilot's Associate, have requirements for 'real-time' processing, experiments must be run to examine the possibilities for speed up. The results of a parallel branch and bound test can be used as a benchmark for further research as well. Since all problems used in this research could not be examined on a dimension-0 hypercube, the performance will be evaluated against a VAX 11/785.

3- Evaluate the Hypercube architecture as a suitable architecture for combinatorial (i.e. search) algorithms.

In conjunction with the development of a good parallel algorithm, the speed up of a problem can be limited by the parallel computer architecture as well. Therefore, the parallel architecture must be evaluated.

4- Identify extensions to this research.

The scope of the research is constrained by time and computing resources. Therefore, recommendations for continued research is essential.

Classes of Search

As pointed out in earlier, typical problems involved in SDI and PA research are categorized as search intensive. Such problems are identified by large, sometimes complex solution spaces and exponential computation time. (See Chapter IV, Parallel Branch and Bound, for definitions and details of how a search is represented.) It is important to keep this research as general as possible. Therefore, the general form of any state space search, branch and bound, is used for the performance experiments. This search technique is important to SDIO and other military problems where optimal solutions are necessary. The basic idea behind a search is to accept a problem in its initial state and 'search' the solution space for a goal state. As a search progresses, a path through the solution space is created. While building this path, choices on which path to follow are made. These choices are the 'branching' part of branch and bound. Other times, it is possible to determine that an answer does not exist down a specific path. This is the 'bound' portion of branch and bound. Using a general search definition is useful during the analysis and design phases of the

research, but specific problems must be used for actual performance measures. To this end, two specific categories or classes of search are defined for this research.

The first search category is called 'backtracking' search. The ranking of all points in the solution space are set to the same value. One reason for such an approach is that an accurate ranking function can not be computed for the problem. Problems that are classified as 'backtracking' search show true exponential computation complexity. The N-queens problem is used in this research as a typical backtracking search. The problem is conceptually quite simple. Given an N-by-N playing board, place N chess queens such that no queens are attacking. This problem is characterized by an exponential time complexity, $O(2^n)$, to find all solutions. Even though the N-queens search does not use a cost function, a bound function is used to eliminate the needless search of some parts of the solution space.

The second class of search is called least-cost branch and bound. This class of problems uses the ranking function and bound function to guide the search. As the name implies, the search progresses based on the 'least-cost' path. Deadline job scheduling is used in this research as the typical least-cost branch and bound problem. The problem is defined as follows. Given a set of jobs, with each job defined by the 3-tuple (p_i,d_i,t_i), where

p_i = Penalty for not scheduling job i
d_i = Deadline by which job i must be completed
t_i = Time to run the job i

Find the largest subset of the jobs that can be run by their deadline while minimizing the total penalty. This search uses both the ranking function to identify potentially good solution paths and two bound functions to eliminate needless searching in parts of the solution space.

While these problems are computationally trivial as compared to the problems faced by SDIO and PA researchers, it must be emphasized this research is investigating the performance improvements of the class of search problems on a parallel computer. Hence, the N-queens problem and the deadline job scheduling problem should be evaluated on there ability to categorize the two classes of search used during the performance experiments.

Parallel Processing Issues

Before moving on, two fundamental issues of parallel processing must be addressed. These issues, maximum parallelism and parallel design, form a basic set of constraints for any type of parallel problem solving. First, the concept of maximum parallelism describes inherent performance limits within a problem, an algorithm, or a parallel computer architecture. While some of these limits can be circumvented, others become the 'bottleneck' of performance. Second, the concept of a formal parallel design methodology has yet to emerge from the research. The goal here is to find a way to map a problem into a parallel computer architecture while at the same time extracting as much parallelism from the problem as possible.

Maximum Parallelism. Simply stated, the concept of maximum parallelism places a constraint on parallel problem solving. This constraint may take several forms. First, the problem may inherently have limitations that cannot be overcome. Second, a poor algorithm may inhibit parallel activity. Finally, parallel computers focus on solving specific classes of problems. The following examples should reinforce the concepts of maximum parallelism and the impact of these limitations on the solution.

Problem Limitations. In his thesis, Norman uses this simple example to show the limits of parallelism within a problem (21:16-22). Given the Lisp expression, CONS(A,B), what is the maximum speed up? In Lisp, the function CONS simply constructs a list with the elements of A as the first elements followed by the elements of B. During the execution of the CONS expression, two operations are performed. First, A and B are evaluated. Second, a pointer is assigned to construct the new list. If the actual performance of the CONS is taken to be negligible (assigning a pointer), then the maximum speed up is dependent on the evaluation of A and B. If the two symbols have approximately the same time for evaluation, then the largest possible speed up from performing the CONS in parallel is only 2x (two times). This problem creates an inherent performance limit that cannot be overcome. The Lisp CONS example is a bit misleading. The evaluation of A and B was assumed to be independent. If one parameter relied on the other or if one parameter required more evaluation time than the other, at some point, one process would be blocked while waiting for the other process to finish. This would immediately begin to reduce the ideal 2x (two time) speed up. Therefore, dependencies among subproblems also lead to decreases in the maximum parallelism of a problem.

Algorithm Limitations. As shown above, the advantages of parallelism is highly dependent on the problem being solved. If any relationships among subproblems exist, then additional performance limits are imposed on the parallel solution. Besides limitations inherent to the problem, a poor algorithm could limit parallel performance. For example, in Akl's simulation of the parallel AND-OR Tree Search, he demonstrated such a phenomenon. Independent of the branching factor of the tree, the depth of the tree, and the probability distribution of the random number generators at the terminal nodes, he showed that the maximum parallelism occurred at approximately 5 nodes (1:197-199). Such a constraint may be overcome if the performance limit is due to a poor algorithm design.

Architecture Limitations. Parallel computer architectures tend to focus on specific classes of problems. Probably the most famous supercomputer, the Cray, is an array or vector processor. The Cray would not be suited for this research into parallel search algorithms. Therefore, as part of analysis and design, the problem must be matched with a compatible parallel processor. Without taking this into consideration, performance degradation is guaranteed.

Parallel Design. The second parallel processing issue deals with mapping a problem into a parallel solution. For humans, thinking in parallel does not come naturally. Therefore, a design methodology is needed to describe a problem such that parallel activity can be identified. Since prior research has not defined a parallel design strategy, one goal of this research is to investigate a formal approach to parallel design. One issue of a parallel design is the concept of 'granularity.' At one end of the scale, fine-grained parallel activity focuses on several small tasks. Parallel programming languages are an example of a fine-grained approach to parallelism. At the opposite end of the scale, coarse grained problems are divided into only a few tasks because of subproblem dependencies or because of the inability to identify parallel activity. The granularity of the design is also dependent on the parallel architecture that executes the solution. While some parallel computers can handle the fine grained approach others can only be used for course grained implementations.

Without too much insight, one can see maximum parallelism and parallel design issues form a dicohotomy. If the solution to the problem is not designed correctly, then the resulting algorithms cause performance bottlenecks. On the other hand, a good parallel design could have terrible performance when placed on an inappropriate parallel architecture. These issues are central themes in this research.

Overview of the Thesis

In summary, this completes the first part of the introduction to this thesis. The problem and goals of the research are straightforward. But, as the analysis shows the final parallel solution may be limited by constraints inherent to the problem, particular to the parallel design or algorithm, or imposed by the parallel architecture. Hence, the approach to solve the problem and to meet the goals must follow a logical path. First, Chapters I and II present the fundamental concepts and goals of the research as well as details about the Intel iPSC computer. Chapter II completes the introduction with a description of the parallel architecture used in the research. Next, Chapter III analyzes parallel design. Here an examination of three approaches to map a problem into a parallel solution identifies the object-oriented design methodology as a good candidate for parallel design. Chapter III finishes this analysis with a formal approach to conduct an object design. In the third section, Chapters IV and V present a formal object design of parallel branch and bound. Chapter IV defines a general parallel branch and bound design. Based on this general design, Chapter V describes the implementation of the N-queens and the deadline job scheduling problems that will be used in the performance experiments. Finally, the actual performance experiments are defined and the results analyzed in Chapters VI and VII. Chapter VI defines the performance measures used to compare the iPSC with an industry standard, the VAX 11/785. Chapter VII finishes the thesis with conclusions about this research and recommendations for further study.

II. Intel iPSC Hypercube

Chapter I introduced this research with a discussion of the basics of parallel processing, a description of the problem, a definition of the classes of search, and an outline of fundamental parallel processing issues. This chapter concludes the introduction with a description of the hypercube interconnection and the Intel iPSC computer. Basically, the iPSC is a general purpose parallel architecture with its processing elements (nodes) connected in a hypercube topology. First, the iPSC design philosophy is examined. Next, a formal definition of the hypercube interconnection is presented. Finally, the details of the Intel iPSC along with its software development environment is discussed.

iPSC Design Philosophy

Initial research on the hypercube, known as the Cosmic Cube, was conducted by Professor Charles L. Seitz at the California Institute of Technology (8,26). His research is supported by the Department of Energy and Intel Scientific Computers. The fundamental basis of the hypercube computer can be described by the process model of computation (26). (An analysis and comparison of several models of computation can be found in Chapter III, Analysis of Parallel Design). Simply stated, the process model describes the interaction of processes using message passing instead of shared variables (26:22). Using such a model, "a programmer can formulate problems in terms of processes and 'virtual' communication channels between processes" (26:23). The iPSC adheres to the process model of computation in two ways. First, the programmer can define and encapsulate a process on any iPSC node. In fact, several processes can be placed on each iPSC node. Second, the iPSC operating system provides a set of message passing primitives for interprocess communication. Figure 2 shows an example of how several processes communicate using

messages. The processor interconnection strategy used to support this model of computation as well as provided good message passing properties is called the binary n-cube or hypercube.

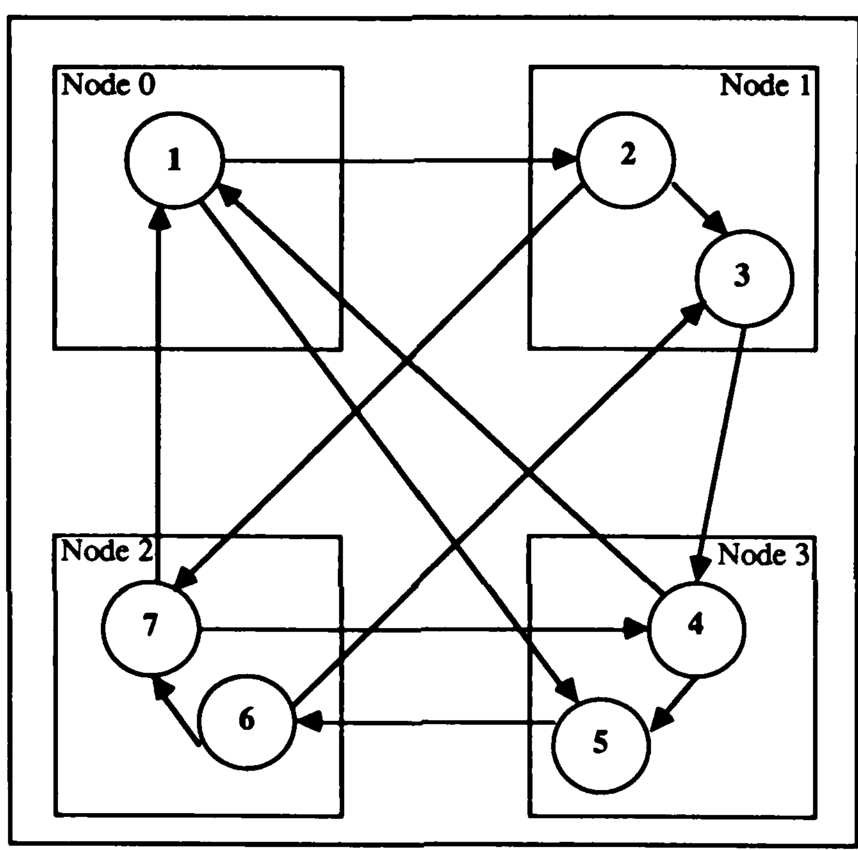


Figure 2: Example of Processes communicating using Message Passing

Hypercube Interconnection

The binary n-cube or hypercube is classified as a limited interconnection strategy since each node is only connected to a few of the total number of processors. As described by Wu, the binary n-cube is a network of 2^n processors where each node has n neighbors (36:239). The number 'n' also describes the dimension of the cube. For example, a dimension-3 cube has 2^3 nodes and each node has 3 neighbors. Each node in a hypercube

can be identified using a binary number of length *n* . Figure 3 shows the labels for a d-3 hypercube. Using the binary node IDs, the nodes of the cube are arranged such that neighboring nodes only differ by a power of two. Tuazon et al shows that the Hamming distance between neighboring nodes is 1 (34:667). He also points out that the Hamming distance can be used to determine the distance between any two nodes in the cube. Figure 4 shows the example of a 3-dimension cube. Notice the node numbers and the Hamming distance between adjacent nodes.

Node	Node Number
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Figure 3: Node Numbers for a 3-dimension cube.

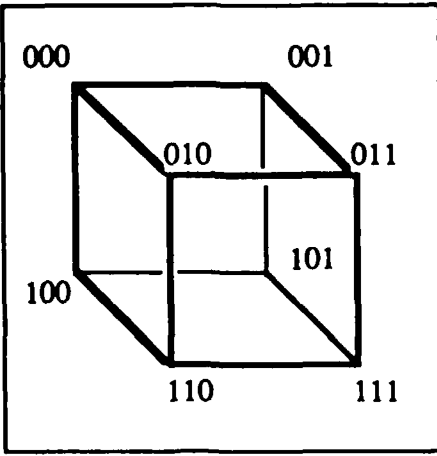


Figure 4: Three-Dimension Cube structure, with vertices labeled from 0 to 7 in binary (28:66)

Since the most efficient message passing occurs between nearest neighbors, it would benefit programmers to use such process/message passing during the problem formulation. Unfortunately, many problems cannot be mapped into a true hypercube communication pattern. But, by defining specific message communications among the nodes, a programmer can configure the hypercube into several logical structures, such as ring, tree, mesh, grids, torus, and bus (17,26). Using these structures, nearest neighbor communications is maintained and the structure of the parallel solution also matches the structure of the problem.

Intel iPSC

The Intel Personnel Supercomputer or iPSC, based on the original work at Caltech, uses the processor/message passing paradigm. The physical configuration of an iPSC has two components, the intermediate host and the hypercube. First, the intermediate host provides a development environment and cube controller for programmers. Intel offers its own 310 microcomputer running the Xenix operating system as the intermediate host. Xenix is a UNIX version 7 derivative with enhancements from Berkeley, Microsoft, and Intel (17). While other machines with an Ethernet interface that run Xenix or Berkeley 4.2BSD UNIX can also be used to connect to the Intel 310, the 310 microcomputer must be used to compile and run the iPSC application.

The second part of an iPSC, the hypercube, is a cabinet or series of cabinets that house the processor boards and communication lines for the hypercube processors. The basic hypercube contains 16 nodes (4-dimension cube) with the largest configuration containing 128 nodes (7-dimension cube). Each node within the iPSC has an Intel 80286 processor and an 80287 numeric co-processor with 512 Kbytes of memory. Each node also has monitor and kernel software. The node monitor contains the 'boot-strap' and self-test routines that

initialize the node at power up. The node kernel serves as an local operating system that performs process management, interprocess communication, buffer management, and memory management.

iPSC Software Development

The iPSC supports four development languages, Fortran, C, Lisp, and 80286 Assembler. For each language an appropriate library of message passing primitives is available. For the intermediate host and iPSC nodes, the following operations can be used,

Host Command	Function
copen	Open a communications channel
cclose	Close a communications channel
recvmsg	Receive a message from the cube
sendmsg	Send a message to the cube
cubedim	Return the dimension of the cube
probemsg	Check for a specific message type
syslog	Write a message to the System Log
load	Load a process into the cube
lkill	Kill a process in the cube
lwait	Place a process in a wait state

Node Command	Function
copen	Open a communications channel
cclose	Close a communications channel
recv	Asynchronous receive message
recvw	Synchronous receive message
send	Asynchronous send message
sendw	Synchronous send message
cubedim	Return the dimension of the cube
syslog	Write a message to the System Log
probe	Check for a specific message type
status	Check the status of a channel
mynode	Return my node number
mypid	Return my process ID number
clock	Return Clock Ticks (1/60th of a sec resolution)

Intel also offers an iPSC simulator that runs under Xenix or 4.2BSD. While the simulator can be used to prototype applications or to learn the basic concepts of the iPSC, it does not model the actual parallel activity of the iPSC. Therefore, the iPSC simulator should not be used to 'load balance' or to 'fine tune' an application.

Conclusions

The Intel iPSC is a flexible, general purpose parallel computer architecture. Created from original research at Caltech, the iPSC offers a true message passing architecture that gives the programmer a wide range of logical interconnection strategies. With the introduction to this thesis complete, it is time to focus on the problem of measuring the performance of parallel branch and bound search. Before such tests can be executed, a parallel design of the sequential branch and bound technique must be created. Because a formal parallel design methodology does not exist, this research investigated strategies to map a problem into a parallel solution. The next chapter analyzes these strategies.

III. Analysis of Parallel Design

In the previous two chapters, the classes of problems, the issues of parallel processing, and the description of the Intel iPSC Hypercube computer established the scope and fundamental concepts of this thesis. At this point, the research focuses on the development of a parallel design methodology. This is the first goal of the thesis. With a design approach in hand, then a formal development of the N-queens and deadline job scheduling can be presented. During the analysis in this chapter, it is important to remember that the ultimate goal is to speed up the performance of the branch & bound search using a parallel computer. Therefore, the analysis and design decisions are influenced by the nature of search problems as well as the characteristics of the iPSC computer. These same decisions may or may not be appropriate for sequential problem solving or other parallel computer architectures. Because of time limitations, this research cannot develop and prove a new design methodology. Therefore, the analysis is restricted to three techniques used today, abstract data types, traditional design methodologies, and models of computation.

Parallel Abstract Data Types

This section examines the concepts of abstract data types for parallel design. The study of abstract data types (ADT) or data structures is a fundamental course in most computer science and computer engineering curricula. Horowitz and Sahni developed a formal definition of ADTs along with techniques to analyze computer programs (15). Extending their ideas, a formal description of parallel abstract data types can be developed. But, as the analysis shows, this approach only exploits limited parallelism of a problem.

Abstract Data Type. Using the Horowitz and Sahni definition, a data structure or abstract data type (ADT) can be described by a set of Domains, a set of Functions, and a set of Axioms (15:1-7). The set of Domains defines the mapping of the ADT functions. This concept is similar to a math function that maps from one 'domain' into a second 'domain' (20:43). Figure 5 gives an example of an abstract data type defining a Stack. The set of Domains for the Stack are {item, stack, boolean}. Notice how the Stack functions map from one domain to a second domain. Next, the set of Functions define the legal operations for the data type. The Stack ADT (Figure 5) defines five functions, {Create, Add, Delete, Top, Isemts}. Finally, using the set of Axioms, the designer can prove the correctness of the ADT. The axioms show the error conditions and legal combinations of the functions. Many common ADT's used in computer programs include array, queue, stack, string, graph, and tree. Sometimes a programming language abstracts the definition of the data structures. For example, most languages support arrays as a primitive data type. Other data types must be designed, implemented, and proven correct. Therefore, ADTs can be used in a hierarchy to build more and more complex data structures.

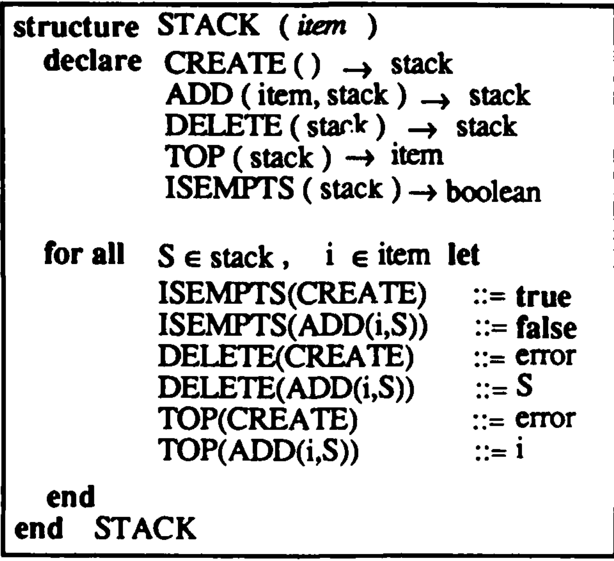


Figure 5: STACK Abstract Data Type (15:67)

Parallel ADT. Following the same paradigm established by Horowitz and Sahni, a parallel abstract data type (PADT) can be defined with a set of Domains, a set of Functions, and a set of Axioms. The set of Domains defines the mapping of the PADT functions. The set of Axioms will be used to prove the correctness of the PADT. Since the set of Domains and the set of Axioms only define the parallel abstract data type, no parallelism can be exploited from them. Therefore, the only parallelism that may exist is in the set of Functions. The functions that can be performed in parallel define 'active' data structures. For example, each element of a linked list could be placed on a separate processor. Any computations on the entire list may be completed in O(1) time. The sequential linked list or 'passive' data structure would require O(N) computation time. An interesting, maybe not so obvious, phenomena occurs when using an 'active' data type. Notice in the simple example of the linked list, the order-of time decreased, $O(1) < O(N)$, but the space requirements reversed. The sequential solution needs only one processor and O(N) Space, but the 'active' data structure needs O(N) processors and O(1) Space. Therefore, Parallel Abstract Data Types trade time efficiency for space efficiency.

Analysis of the PADT. The PADT may be used to reduce the Order-of Time computations for some functions at the expense of increasing the Order-of Space requirements. Using the PADT concept to promote fine grained parallelism in computers may be useful in Parallel Programming Languages. As such, the details of the parallelism is abstracted and the programmer can build even higher level structures to define more complex PADTs. Seitz describes this type of parallelism as a *covert* technique to introduce parallelism using sequential processors (26:25). Some commercial parallel processors use this idea to define abstract parallel data structures. One machine, the CM-1 (Connection Machine 1) runs a parallel implementation of Lisp, CmLisp (Connection Machine Lisp). The parallel structure within CmLisp is called the xector (pronounced zek-tor). A xector is defined by a Domain, a

Range, and a Mapping much in the same way a sequential data structure is described (14:33). For example, a list can be defined as a xector where each element resides on a separate processor. Using a parallel data structure like a xector, Lisp programs may run faster by using the inherent parallelism of list processing functions. Hillis shows some of the CmLisp time order complexity reductions (14:38),

List Operation	Vector	List	Xector
Remove	O(N)	O(N)	O(1)
Sort	O(N log N)	O(N log N)	$O(\log^2 N)$
Length	O(1)	O(N)	O(log N)

Unfortunately, using such a mechanism may not exploit the full potential of parallelism within a problem. Basically, the PADT approach maps a problem defined for a sequential computer into parallel data structures. This mapping may not exploit all of the parallelism within the problem itself. Seitz recommends parallelism techniques such as these within each node of the parallel processor because at each node "we are tied to sequential program representations" (26:26). Therefore, to extract the "most" parallelism from problems, another approach must be taken.

Design Methodologies

In the previous section, the parallel abstract data types only provided a limited parallel solution because the PADT does not exploit the parallelism of the problem itself. Therefore, a general approach is needed that examines the problem for potential parallel activity. Such approaches are design methodologies. Since a parallel design methodology does not exist, an analysis of design strategies is in order. After reviewing three popular methodologies, a design style for this research was selected. While analyzing design strategies, it is important to remember the ultimate goal of design is to accurately represent or model the problem space

(5:39). If a methodology can do this, then the designer should be able to identify the inherent parallelism within the problem.

Traditional Design. Software design methodologies are used to define a disciplined approach to problem solving. Figure 6 shows a functional representation of a problem, a design methodology, and a solution. The problem is characterized by real-world objects, operations, algorithms, and results (5:38-39).

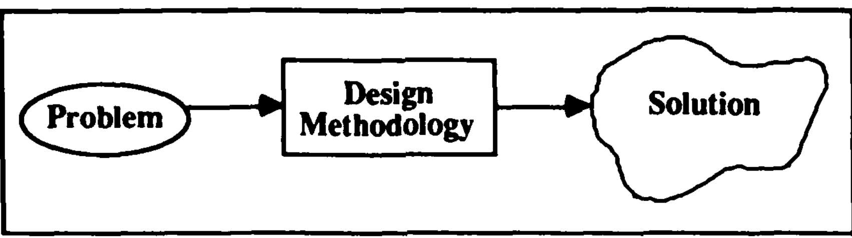


Figure 6: Functionality of a Design Methodology

The design methodology creates a bridge from the real-world or problem space into the solution space. The solution for this discussion is modeled by computer hardware and software. Three popular design methodologies include, Top-Down Structured Design, Data-Structure Design, and Object-Oriented Design (4,5,12,23,37). The scope of this research does not permit an in-depth review of each of these design styles, but a summary of each methodology may help the reader.

Top-Down Structured Design. Top-Down design is one of the more popular design styles since it is well suited for several programming languages and VonNeumann computer architectures. Simply stated, Top-Down Structured Design is a method where a system is decomposed into its major functions. Those functions are decomposed into smaller pieces, and so on until a function can be described in a programming language (37:2). While this

design methodology works well for sequential machines, it does not support features needed when describing a parallel solution, such as distributed control and interprocess communication (5:32).

Data-Structure Design. This design methodology originated from the COBOL language. Based on work by Jackson, Data-Structure Design defines the data structures of the system. From the data structures, the programmer defines the structure of the program modules (5:32). This design style is similar to the abstract data type approach to parallelism. Data-Structure Design has the limitations of the abstract data type approach as well as limitations on describing the solution space of the problem.

Object-Oriented Design. Starting with work at MIT, Stanford, and the University of North Carolina, object-oriented design is a flexible design methodology. Used not only as the basis for programming languages, object-oriented design has also been used in other disciplines. The Intel 432 microprocessor is an example of an object-oriented computer architecture. The object-oriented design strategy is based on the concept of decomposing a problem into objects, operations, and interactions among the objects.

Analysis of Design Methodologies. No one has developed a methodology for parallel design. Therefore, one must rely upon a software design style that models the problem accurately as well as provides a flexible design environment. The results of such a design should help the programmer identify the parallel parts of the problem. Commonly used design techniques, such as Top-Down Structured Design and Data-Structure Design, cannot accurately define the problem space and therefore do not support parallel design or parallel processor implementation. On the other hand, the object-oriented design methodology meets these requirements. Some work with object-oriented parallel design has already been carried

out by researchers using Occam, the concurrent language for Inmos' transputer (3). Booch also points out that from his experience "an object view of the problem space lends itself well to exploiting massive parallelism" (3). Because of the ability to accurately describe the solution space of a problem, this research will be based on the object-oriented design methodology.

Parallel Models of Computation

An analysis of parallel models of computation reinforces the decision to use the object-oriented design methodology. A look at these models also reviews the basic concepts of the Hypercube architecture. (See Chapter II, Intel iPSC Hypercube, for details). Just as a design methodology is used for a disciplined approach to problem solving, a model of computation establishes a formal definition of a computer system. Figure 7 shows the relationship between a model of computation and computer architectures.

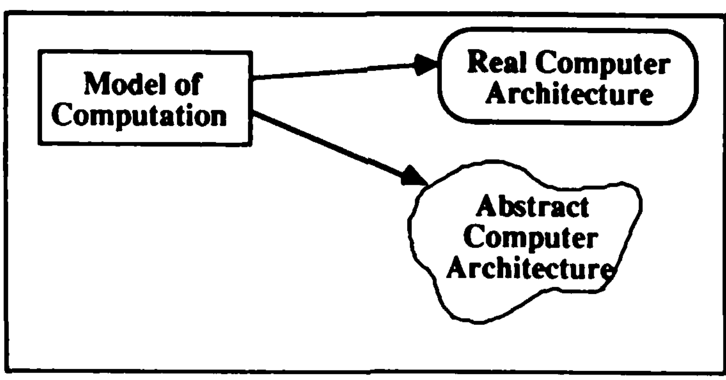


Figure 7: Model of Computation

As the figure implies, models of computation are used to define computer architectures, either real or abstract. Gajski and Peir identify three basic sequential models of computation (11:10-11), (1) Operational Model, (2) Applicative Model, and (3)VonNeumann Model. The operational model, such as a Turing Machine, is used as a simple and concise mathematical

description of a computer system (11:10). A slightly more complex model, the applicative model, also has a concise description but is not supported by a mathematical definition (11:11). A pure Lisp Machine can be described by an applicative model of computation. Finally, the VonNeumann model serves as the basis for almost all conventional computer systems. Computers based on the VonNeumann model are much more complex than their operational or applicative counterparts.

A goal of this research is to explore parallel design strategies. As part of that goal, this research selected a design strategy and a model of computation that are compatible with each other. For example, the VonNeumann model along with the Top-Down Structured Design Methodology seem to work well for most sequential machines. Unfortunately, as Treleaven points out, the sequential models cannot adequately describe parallel computers (33:275). Therefore, they cannot be used in this research. In response to this, parallel models of computation have been developed. Gajski and Peir describe a parallel model of computation as a graph, where the nodes of the graph define the tasks or processes that must be carried out, and the arcs of the graph define the order of node firing. Despite this rather simple idea about parallel computers, several complex parallel computer architecture have been designed. Three specific parallel models of computation include the Data-Flow Model, the Control-Flow Model, and the Process Model (11,26,33). The scope of this research does not permit an in-depth review of each model of computation, but a summary may help the reader.

Data-Flow Model of Computation. In the data-flow model, each node may 'fire' when all of the data it needs for execution is available. Each node receives its data and forwards results along the arcs using a 'token'. Each token contains data or instructions for a node. See Figure 8 for an example of the data-flow computation. An important aspect of data-flow machines is their ability to extract the parallelism of a problem at run-time instead of at design time.

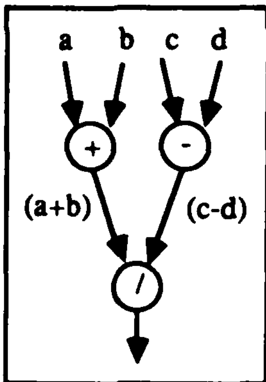


Figure 8: Execution of a data flow computation (a+b)/(c-d) (11:12(a))

Control-Flow Model of Computation. The control-flow model uses tokens to pass pointer information between nodes. The actual data resides in a global memory. This scheme is useful when performing computations on large data structures like matrices. Figure 9 shows the computation of (A+B)/(C-D) using a control flow graph.

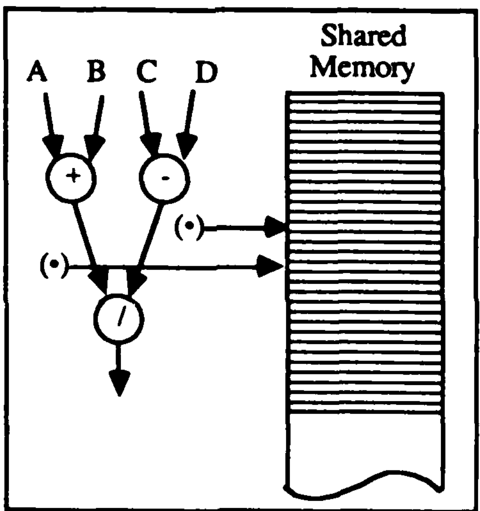


Figure 9: Execution of a control flow computation (A+B)/(C-D) (11:12)

Analysis of the Data-Flow and Control-Flow Models. The Data-Flow and the Control-Flow Models of Computation are useful in describing a parallel solution to specific problems. Unfortunately, both of these models limit the definition of a system by restricting the class of parallel processors. For example, the data-flow model maps the problem to data-flow computer architectures and the control-flow model maps the problem to parallel computers with a shared memory. A general parallel model of computation is needed to remove the limitations of a specific architecture while designing a parallel solution. If a programmer has the freedom to choose a parallel computer, then upon completion of the general design, a suitable parallel architecture can be selected. On the other hand, if a programmer is restricted by a specific computer architecture, a general model of computation can still be used to describe the problem. One such model with this flexibility is the process model of computation.

Process Model of Computation. The process model of computation describes the interaction of processes using *message passing* instead of shared variables (26:22). Both the data-flow and the control-flow base their computer descriptions on shared data. Using the process model, a programmer defines a general parallel solution by formulating the problem in terms of processes and "virtual" communication channels between processes. Interestingly, the object-oriented design methodology decomposes problems based on objects (independent processes) and communications or message-passing among those objects. Because the Hypercube is defined by the process model of computation, parallel solutions map naturally to the iPSC Hypercube using the object-oriented design style. Seitz also recommends the process model approach to parallel design because (26:26),

We do not know how to write a program that translates application programs represented by old, dusty FORTRAN decks into programs that exploit concurrency between nodes. In fact, because efficient concurrent algorithms may be quite different from their sequential counterparts, we regard such a translation as implausible, and instead try to formulate and express a computation explicitly in terms of a collection of communicating concurrent processes.

Object-Oriented Design

In summary, the object-oriented design methodology is used as the parallel design strategy for this thesis. The object model not only represents the problem space accurately but also maps naturally into an iPSC implementation. At this point, a description of object design is in order. This discussion also includes a design approached defined by Booch (4,5). In short, the object-oriented design methodology can be defined as a software design tool "in which the decomposition of a system is based on the concept of an object" (4:211), and an object can be defined as an instance of an abstraction from the problem space. Objects typically initiate action or respond to requests from other objects. Some computer languages like Smalltalk and Lisp Flavors directly support the definition and interaction of objects. Object design is not restricted to computer languages. Besides a general software design style, object-oriented design has also been used for computer architectures, such as the Intel 432. A problem can be accurately abstracted into the solution space using only four basic constructs, (1) Class; (2) Method; (3) Message; and (4) Object.

Class. A class defines a template for an object. This template includes a set of attributes about an object and a set of operations for that object. Classes may form a hierarchy in which each subclass inherits the attributes and operations of its ancestors. A hierarchy of classes is generally used to "factor the common properties of a set of objects" (4:216). Figure 10 shows a class hierarchy of the class of Aircraft, Jet, Piston, and Turbo-Prop.

While the specific attributes and operations are collected in the Jet, Piston, and Turbo-Prop Classes, the common properties are identified at the highest (most general) class, Aircraft. For example, if this hierarchy is used within an air traffic control system, the attribute 'position' pertains to all aircraft. Hence, the attribute 'position' is associated with the most general class, Aircraft Class. On the other hand, specific attributes about a Learjet must be incorporated into the Jet Class.

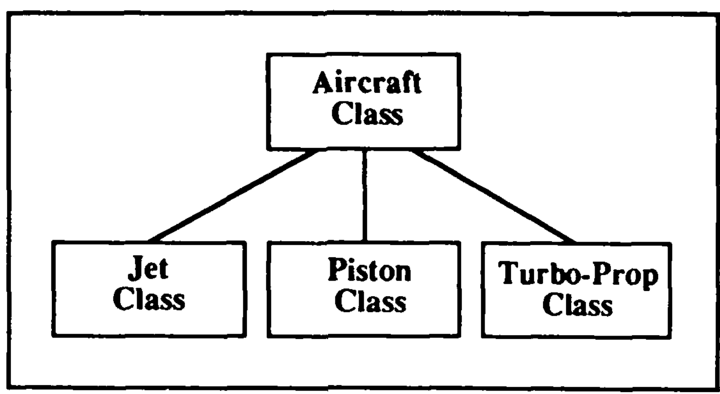


Figure 10: Example Class Hierarchy

Message. Objects use messages to initiate an action or a response to another object. The response could range from changing the state/attribute of the object to sending messages to other objects. The message passing paradigm creates two important concepts, interface specification and information hiding. As an interface specification, messages serve as the rules-of-communications among objects. The interface specifications may be reguarded as the external view or the abstract behavior of an object (4:217). The second concept, information hiding, protects the details of how the operation takes place. For example, using the Aircraft Class Hierarchy once again, one may "send" the message **TYPE** to an aircraft object. As an interface specification, the response should be an appropriate aircraft type such as B747, DC-10, etc. To re-enforce the concept of information hiding, we have no idea how the **TYPE** message was actually calculated.

Method. A method "denotes the response of an object to a message from another object" (4:215). The response may change the state of an object, return a value, compute a function, or send messages to other objects. Methods define the internal behavior of an object. As such, the method explicitly defines the functions, parameters involved, data structures, and details of how the response is calculated. Continuing the example, the **TYPE** message could be an attribute of the Aircraft Class. Then, the response would only consist of returning the value of that attribute.

Object. An object is a unique instance of a class. The object inherits all the attributes and operations of its parent class and its ancestors. An example of an object would be United Flight 300. The attributes of United 300 include Type = B747 , Position = 48'North 100' West, Ground Speed = 600 Knots, Destination = LAX, etc.

Using the above building blocks, the object model can represent any abstraction from the problem space. This includes abstract data types, entities within the problem space, interactions among the entities, and object support tools such as complex data structures. Since the concept of an object encompasses many abstractions, Booch tries to simplify the idea a bit more by identifying three subclasses of objects, the Actor, the Server, and the Agent (4:216). While Actor objects initiate action from other objects, Server objects only respond to requests from other objects. Finally, Agent objects combine the properties of the Actor and the Server by initiating action and responding to requests. For example, the control within the problem can be classified an Actor object since it initiates action; and a queue data structure is classified as a Server object since it only responds to requests.

Object Design Approach

Even though most computer languages do not support object-oriented concepts directly, an object design can be useful in almost any software development environment. Since most computer languages support subroutine, function, or procedure calls, programmers can build abstract data types. As Booch points out, abstract data types represent Agent objects and Server objects (4:216). Therefore, one must find a representation of an Actor object to complete the transition from object-oriented languages to other computer languages. Typically, Actor objects are the control or main module of the procedural language. Therefore, the Actor object becomes the control or finite state machine within the computer language. Even though object design concepts create a good model for almost any problem, a well defined approach is necessary. Booch recommends the following steps (4:213-214, 5:38-44),

1- Define the Problem
2- Identify Objects and Attributes
3- Identify Operations
4- Establish Visibility
5- Establish Interface
6- Implement the Objects

Define the Problem. Before a good design begins, it is necessary to understand the problem thoroughly. Sometimes the problem can be examined using an analysis technique such as dataflow diagrams. Other times, an analysis technique must be coupled with a software prototype to aid in problem understanding.

Identify Objects and Attributes. At this point, identify all of the Actors, Agents, and Servers within the system. Typically, nouns identify the objects. The attributes of an objects are the necessary pieces of information that an object needs to do its job.

Identify Operations. Next, determine all of the operations performed by the object or on the object. During this phase, any timing considerations must be identified as well. For example, a 'window' object in a computer terminal window-system must be 'open' before any operations can be applied.

Establish Visibility. At this point, the interactions and dependencies among the objects are identified. This step builds the topology of the objects and helps us identify the mapping of the problem space into the solution space.

Establish Interface. Next, the interface or external view of each object is created. This interface sets up the rules of communications between objects.

Implement the Objects. Finally, the internal representation of each object is established.

The internal view defines the data structures and details of how the objects performs its functions. Many times objects are implemented from lower level objects. Systems developers can also experiment with different implementations as long as they obey the interface specs.

The figure on the next page shows the mapping of the four object-oriented constructs into the Ada, C, and Pascal languages (see Figure 11). Ada supports all of the features of an object-oriented language except for the class hierarchy inheritance (4:217). Using a bit more care, Pascal and C Language may also be used for object-oriented programming. C Language is used for all experiments in this research.

	Class	Methods	Objects	Messages
Ada	- Package with private or limited private types.	- Subprograms exported from package spec.	- Instances of private or limited private types - State Machine (Actor) - Tasks/Task Types (Actor)	- External View supported by separate pkg specification - Internal View supported by separate pkg body
C	- Abstract Data Type - Primitive Data Type	- Functions	- Instances of an Abstract Data Type or Primitive Data Type - State Machine (Actor)	- Interface to Functions - External View Supported by "Header Files" - Internal View Supported by function body
Pascal	- Abstract Data Type - Primitive Data Type	- Procedures - Functions	- Instances of an Abstract Data Type or Primitive Data Type - State Machine (Actor)	- Interface to Functions and Procedures

Figure 11: High Level Language/Object Constructs Matrix

Conclusions

Many researchers have developed ways to speed up computations using parallel techniques. Some have focused on parallel data types while others re-evaluate each problem for inherent parallel activity. One goal of this research is to investigate an approach for parallel design. This analysis selected the object-oriented design methodology because of the ability to accurately model the problem space as well as map the final design into the hypercube implementation. For this research, the six-step object design approach as defined by Booch is used. Even though the steps are presented sequentially, anyone using them should expect to visit each step several times before declaring the design final. This chapter concludes the analysis of parallel design techniques. Continuing, the research proceeds with a general parallel branch and bound design based on the object model in the next chapter.

IV. Parallel Branch and Bound

Using the six step object-oriented design approach defined in the previous chapter, this chapter presents a formal object design of parallel branch and bound. A general design is given and some insight about the design decisions are examined for each step. The implementation details of the N-queens problem and deadline job scheduling can be found in the next chapter. While this design is presented in a orderly fashion, the actual design involved an interative approach. The designer should expect to visit each step several times. The design in this chapter proved useful in two ways for this research; (1) the design of an actual problem allows this research to draw conclusions about the usefulness of the object-oriented design methodology for parallel processing; and (2) the design prepares the problems for a performance test.

Define the Problem

Before the design is presented, the concept of search must be defined. Search is a basic Operations Research (OR) and Artificial Intelligence (AI) programming technique. Such a strategy is used when problems cannot be solved using direct methods (i.e. formulas, algorithms, etc) (25:55). Typically, a search is reguarded as a sequential, centralized control strategy that accepts a problem and its initial state and 'searches' the solution space for a goal state. The efficiency of a search is dependent on how well it uses the domain specific knowledge of the problem (25:55). Several specific search strategies have been developed. Each strategy varies the way the solution space is examined. While sometimes the entire solution space is blindly searched for an answer, other search techniques use heuristics or rules as a guide through the solution space. Below, Rich lists examples of specialized search strategies (see Figure 12).

Branch & Bound	**B***
Depth-First Search	**Breadth-First Search**
Best-First Search	**AO***
A*	**Heuristic Search**
Hill Climbing	**Alpha-Beta Cutoff**
Constraint Satisfaction	

Figure 12: Search Strategies (25)

From this basic introduction, the details of the search programming technique is in order. The following definitions, descriptions as well as examples should prepare the reader for the search experiments used in this thesis. For more information, consult (2) (16), and (25).

First, the problem space for a search is typically represented using a tree or network organization (a network can be represented as a tree) (16:325). Horowitz and Sahni describe the search tree as follows (16:325-329): The root of the tree represents the *initial state* of the problem (see Figure 13). Each nonterminal node in the tree represents a *problem state* in the search. The set of all paths from the root node to any node in the tree define the *solution space* for a given problem. As the search progresses, a node that has been inspected but all of its children have not been generated is called a *live* node. The live node currently being expanded is called the *E-node*. Finally, a *dead* node is one that has been inspected and all of its children have been generated.

Even though trees are used to represent the solution space of a search, the tree is usually not stored explicitly in the computer. Because search problems have the additional overhead of combinatorial explosion due to the branching factor or the depth of the tree, only the portions of the tree needed are kept in storage.

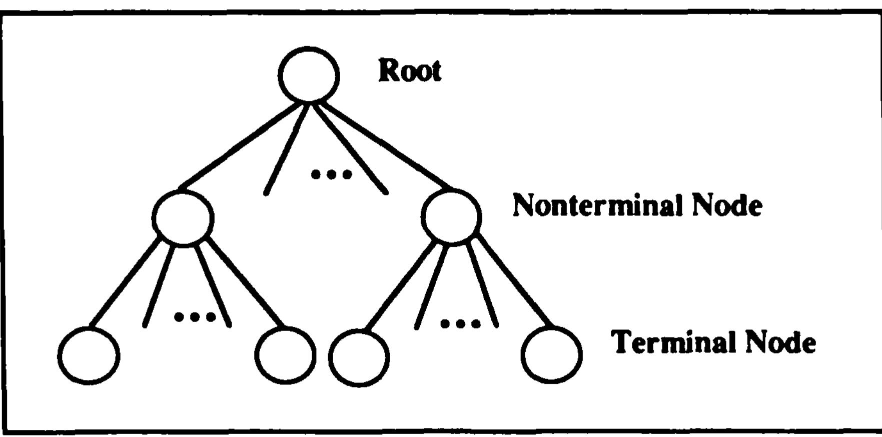


Figure 13: Search Tree with Node Definitions

Using the definitions of the search tree, a blind search, such as depth-first and breadth first search, does not use knowledge of the problem domain to control the search process. Given the tree in Figure 14, a depth-first search visits the nodes in the following order,

A - B - D - E - C - F - G

and a breadth-first search visits the nodes in this order,

A - B - C - D - E - F - G

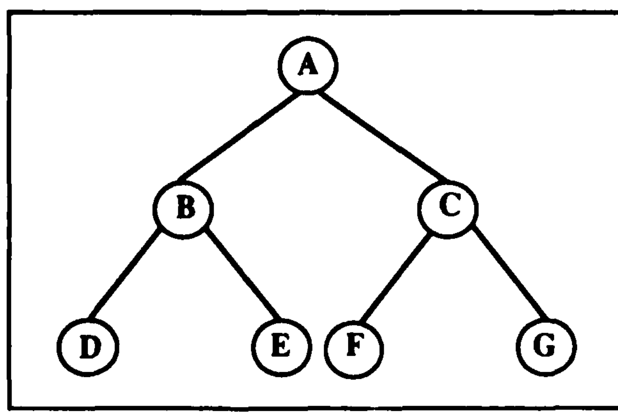


Figure 14: Example Search Tree

The other techniques, called intelligent search, try to narrow the search space, shorten the search time, and reduce the storage needed by applying knowledge of the problem domain to control the search. The following actions are used to meet the three goals of 'intelligent' search (2:59),

1- Decide which node to expand next.
2- Select the most promising successors when expanding a node.
3- Eliminating or pruning the search tree.

The most general search strategy that incorporates all three of these techniques is known as branch and bound search. Because of its general form, branch and bound can model 'blind' as well as 'intelligent' search by changing certain parameters (see next section).

<u>Branch and Bound Search</u>. Branch and bound search is the general form of any state space search. Therefore, many of the search strategies used in OR and AI can be modeled using branch and bound. Furthermore, it is important for research of this type to attack the most general form of a problem. Branch and bound meets this requirement.

Branch and bound search is characterized by a ranking function, c(X), and a bounding function, b(X). First, the ranking function measures the cost to reach an answer from node X in the search tree. During the search, the E-node (next node to expand) is selected from a list of live nodes. The E-node selection may be based on the value of c(X). For example, if c(X) is the cost to reach an answer and the E-node selection picks the node with the least c(X), then the branch and bound search models a least-cost search. Even though the search follows the lowest cost path to the solution, using such a cost function can be computationally expensive. Computing the cost to reach an answer node usually entails an additional search of a subtree. Therefore, the branch and bound ranking function, c(X), is a trade off

between the time to compute the cost and the efficiency of the cost function. Instead of computing the actual cost to an answer, most branch and bound searches use a function, g(X), that estimates the cost of reaching an answer node. Using g(X), the live nodes are ordered by the following function,

$$c'(X) = h(X) + g(X)$$

where,

c′(X) = Total estimated cost
h(X) = Cost of reaching node X from the root node
g(X) = Estimated cost of reaching a solution from node X

Some examples of using the estimating cost function are (16:372).

$$\begin{array}{l} h(X) = \text{level of node X in the search tree} \\ g(X) = 0 \\ \hline c'(X) = \text{Breadth-First-Search} \end{array}$$

$$\begin{array}{l} h(X) = 0 \\ g(X) \geq g(Y),\ Y = \text{child of X} \\ \hline c'(X) = \text{Depth-First-Search} \end{array}$$

Next, the bound function, b(X), examines node X for specific boundary conditions. If the node passes the bound function, then it becomes a live node. If the node does not pass the bound function, then the node becomes a dead node. The bound function prunes the search tree and therefore eliminates needless computation in parts of the search space where solutions are 'known' not to exist. In fact, some searches may incorporate several bound functions.

Any search that incorporates a cost function for selecting the next E-node and a bound function to prune the search tree is called a branch and bound search. The efficiency of such a search is keyed upon the accuracy of the cost function and the bound function. A poor cost function may direct the search to the wrong part of the search space, and a poor bound function may prune a subtree that has an answer node.

To summarize, search is used when an algorithmic solution does not exist. The search programming technique offers a way to examine all possible points in the solution space for an answer. Because of the exponential nature of a search problem, an exhaustive examination of the solution space is computationally prohibitive. Hence, a variety of search techniques have been developed to reduce the time to find an answer. Sometimes a 'blind' search is appropriate while other times more complex 'intelligent' searches are needed. In any case, the most general state space search is called branch and bound. This programming technique is characterized by two functions, a cost function and a bound function.

Based on this analysis, a dataflow diagram of branch and bound is shown on the next page (see Figure 15). In addition to this analysis, sequential versions of the N-queens and deadline job scheduling were programmed to understand the two problems used in this research. Descriptions of the sequential code can be found in Appendix B and D respectively.

Report Generator

Solution Vector(s)

Terminate Check

Add Live Node to Q

Solve Next Problem

Report

Live Node

Solution

Report

Live Node Q

Live Node

User

Controller

E-node

Select next E-node

E-node

Live Node

E-node

Problem Solver

Live Node(s)

E-node Expander

Children of E-node

Bound Check

Figure 15: Branch & Bound Dataflow Diagram

Before moving to the next step in the object-oriented design, the representation of a node in the search space must be described. For this research, a node in the branch and bound search problem is represented as a vector, $(x_1, x_2, \ldots, x_n)$ (16:323). Each x_i is constrained by explicit and implicit constraints. First, the explicit constraints define the range of values that each x_i can be assigned. For example, $x_i \in \{0,1\}$ is a set of explicit constraints. The x_i's can have the same set or different sets of explicit constraints. All nodes in the solution space satisfy the explicit constraints. The second set of constraints, called implicit constraints, define the relationships among the x_i's. The bound function insures that the implicit constraints are met. Any node in the solution space that meets both the explicit and implicit constraints is an answer node.

Identify Objects and Attributes

Using the dataflow diagram (Figure 15), all objects in the problem domain as well as the attributes for each object can be defined. Branch and bound search has the following objects.

Object	Attributes
Problem Solver	Solution Vector Solution Explicit Constraints Solution Implicit Constraints
E-Node Expander	Solution Vector Solution Explicit Constraints
Bound Check	Solution Vector Solution Implicit Constraints
Termination Check	Terminate Condition
Controller	Current State of the Machine List of Live Nodes

Identify Operations

Next, determine the operations performed by the object and required of the object. During this design step, the designer identifies timing considerations as well. This timing information is vital to identify the parallel and non-parallel parts of the problem. First, examine the operations performed by the objects. Booch points out that these operations roughly match the data flows into an object (4:219).

Object	Operation Performed by the Object
Problem Solver	Find all Children of an E-Node Examine Live Nodes for Solutions and Answers
E-Node Expander	Generate the children of an E-node
Bound Check	Check the Children of an E-Node against the Explicit Constraint Function
Termination Check	Check if solution meets terminate condition
Controller	Solve the Problem Collect Live Nodes Collect Answers

Notice the Controller does not distinguish between a Problem given by the user and expanding the next E-Node for a Solution. Since the User's Problem as well as the E-Node can be any valid state space vector, the Controller does not have to make the distinction.

Next, the operations required 'of the object' are defined. Booch points out that "these operations roughly parallel the action of a data flow from an object" (4:219).

Object	Operation Required of the Object
Problem Solver	Expand E-Node Determine if a Live node = Answer
E-Node Expander	Generate all Children of an E-Node

Object	Operation Required of the Object
Bound Check	Send Live Nodes to Problems Solver
Termination Check	Check for termination
Controller	Solve the Problem Add a new Live Node to the Heap Check for Problem Termination

Before moving to the next step, the designer must identify the timing relationships among the objects.

```
loop
if Problem has Terminated then
    End
else
    Get a Problem to Solve          -- get E-Node
    Solve the Problem               -- expand E-Node
    Check for Bound Conditions
repeat
```

Wah used this timing information to define four branch and bound 'processes' that can run in parallel, (1) Parallel Termination Check; (2) Parallel Problem Selection; (3) Parallel Problem Solution; and (4) Parallel Bound Check (35:96).

Establish Visibility

This step, establish object visibility, defines the topology of the solution space and the structure of the parallel solution. Using a diagram such as Figure 16, the designer identifies the dependencies among the objects. The object model at this point creates a continuum of possibilities for parallelism. At one end of the scale, all objects are placed on one processor. Programmers do this today when writing a sequential program. At the opposite end of the spectrum, each object or a collection of objects are placed on a separate processor. The optimal parallel solution lies somewhere between these extremes. An examination of some visibility diagrams points out how the parallel visibility analysis works.

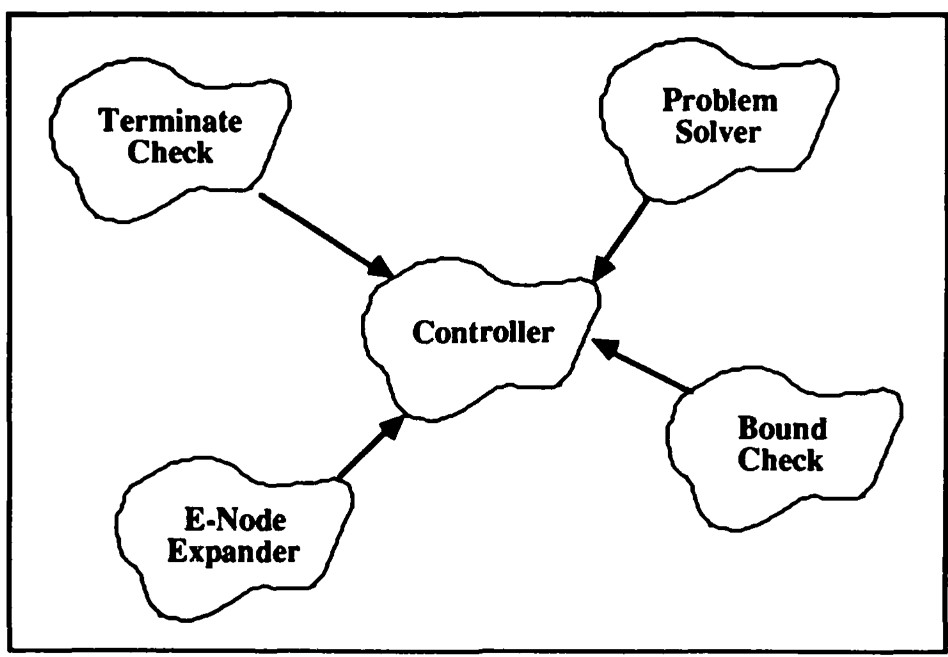


Figure 16: Sequential Branch and Bound Object Visibility Diagram

The first parallel visibility diagram shows a collection of worker processors (Processor #1 ···Processor #N) whose sole purpose is to solve a particular problem (see Figure 17). The remaining objects (controller, bound check, terminate check, and e-node expander) are placed in another processor. Of the four parallel 'processes' defined in the timing analysis by Wah, this visibility strategy attempts to create parallel activity by solving several problems at once. But, this configuration of objects creates a communications bottleneck at the Processor with the controller, e-node expander, bound check, and terminate check. Such a design would limit parallel activity.

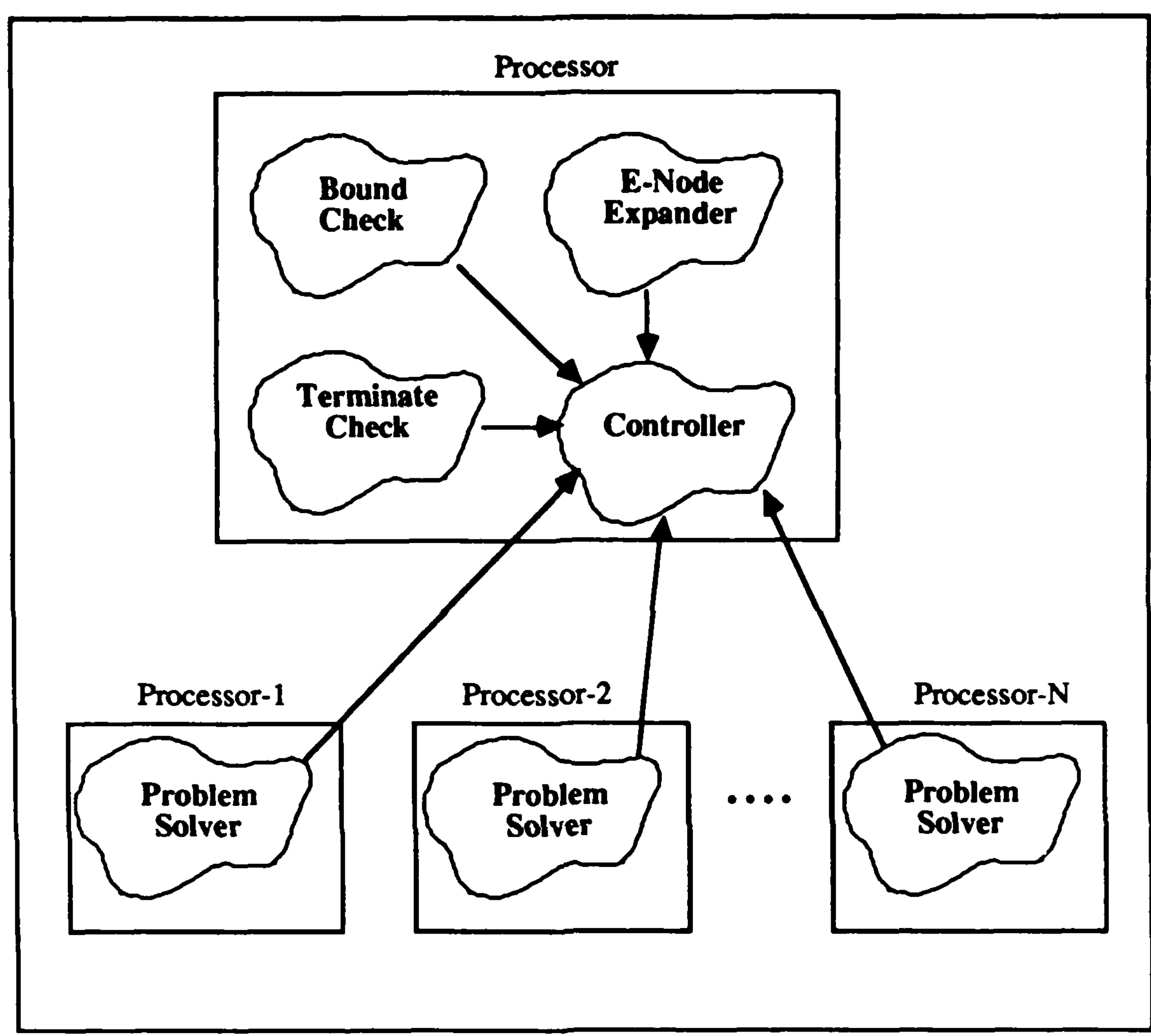


Figure 17: Parallel Object Visibility Diagram #1

In the second parallel visibility diagram, an attempt to increase parallel activity is shown (see Figure 18 on the next page). Here, by distributing complete problem solving objects, the problem solver, the e-node expander, and the bound check, some of the computation workload of the Central Controller is shared. In this formulation of parallel branch and bound, the Central Controller would also be useful in searches that incorporate global boundaries. Unfortunately, the communications bottleneck still exists at the controller. This bottleneck limits parallel activity.

Processor

Terminate Check

Controller

Bound Check

. . .

Processor-1

Problem Solver

E-Node Expander

Bound Check

Processor-N

Problem Solver

E-Node Expander

Bound Check

Figure 18: Parallel Visibility Diagram #2

Finally, the parallel visibility approach used in this thesis is shown in Figure 19. In this parallel design, several instantiations of complete branch and bound search processors are controlled by a Meta-Controller. The Meta-Controller is used to keep Worker Processes busy, maintain the machine state, and terminate the search. This formulation also works well for state space searches with dependencies generated from other branches of the solution space (global dependencies). If a branch and bound search has a global dependency such as an upper or lower bound, then the value of this bound must be available to all branch and bound processors. The Meta-Controller can be used to distribute this bound.

Before leaving this design phase, a mapping from the object-oriented design strategy and the process model of computation can be completed. Notice that the object model describes the problem space as 'fine grained' objects that communicate via message passing. The process model of computation defines a computer system where processes communicate using messages as well. A direct mapping of a process to an object can be made. Or, for efficiency and reduced communications overhead, a collection of objects (on one node) can be consolidated into one process. Therefore, the object model defines the problem space as 'fine grained' and the process model implements the solution as fine or coarse as needed for efficiency.

Figure 19: Parallel Branch and Bound Visibility Diagram used for this research

Establish Interface

Now, the message passing interface is created for each object. This interface describes the external or abstract behavior of the object while the implementation details of how the object performs its operations are hidden (4:214). The interface specifications for the branch and bound search are described below.

Meta-Controller → User
- Accept Initial Problem Vector from User
- Maintain State of the Machine
- Find a Solution to the Initial Problem

Meta-Controller → Controller
- Broadcast Global Upper Bounds
- Distribute problems to solve

Meta-Controller → Terminate Check
- Find all Solutions
- Find the first solution
- Find an optimal solution based on additional constraints

Meta-Controller → Bound Check
- Maintain Global Upper Bound(s)

Controller → Meta-Controller:
- Accept Problem Vector from Meta-Controller
- Find all solution to the Problem Vector

Controller → Problem Solver:
- Given an E-Node return Live Node(s) or Answer Nodes(s).

Controller → E-Node Expander:
- Given an E-Node generate all Children.

Controller → Bound Check
- Given a Child node check against implicit constraints

Controller → Terminate Check:
- Receive a terminate message from the Meta-Controller

From these specifications, the programmer implements a branch and bound search for a specific problem. But, before going to the final step in this design, it is important to understand the importance of the interface. Simply stated, the interface specifications must be defined with parallel operations in mind. The Controller and the E-Node Expander interface is a good example as shown in Figure 18: Parallel Visibility Diagram #2. Assume this interface is defined as follows:

E-node Expander → Controller
- Given an E-node generate one child

And given the following problem state,

Live Nodes	Available Processors
Initial Problem	{1,2,...,N}

no parallel activity could occur. First, the Controller selects Processor #1 and the Initial Problem for expansion. This would leave the state of the machine as,

Live Nodes	Available Processors
<Empty>	{2,...,N}

There is no more work left for the remaining available processors. Once Processor #1 returns from solving the initial problem, the state of the machine will be,

Live Nodes	Available Processors
New Live Node	{1,2,...,N}

Once again, the first processor is tasked to solve the only problem in the Live Node pile. Therefore, this interface specification does not promote parallel activity. With the following interface, parallel activity is much more likely,

E-node Expander → Controller
• Given an E-Node generate all Children.

The sequence of states for the Controller would now look something like this,

Live Nodes	Available Processors
Initial Problem	{1,2,...,N}

Live Nodes	Available Processors
Live Node-1	{2,...,N}
Live Node-2	

Live Nodes	Available Processors
Live Node-k	{...,N-2,N-1,N}
Live Node-k+1	

Live Nodes	Available Processors
Live Node-n-2	{...,N-2,N-1,N}
Live Node-n-1	
Live Node-n	

Now, when Processor #1 becomes available, other processors have been creating more problems to solve. This analysis points out the limitation of the design and the communications bottleneck in the second parallel visibility diagram. A thorough analysis of each interface, such as the one described above, is necessary. A poor interface could limit the maximum parallel activity of a problem.

Implement the Objects

The final step in an object design is to implement the objects. The internal view as well as the details of how the object performs its operations is now defined. Many times a complex object can be implemented from lower level objects thus creating a hierarchy. For example, the Meta-Controller keeps a list of Live Nodes in a queue data structure. This queue is in essence a lower level object with operations and attributes of its own. Systems developers can also experiment with different implementations as long as they obey the interface specifications.

Conclusions

Search is a sequential, centralized control strategy used to systematically examine the solution space of a problem. To map this strategy into a parallel solution, the object-oriented design methodology and the object design approach as defined by Booch was used. In fact, the object design model proved to be quite useful in describing the problem of branch and bound search as well as extracting parallel activity. While this chapter presented a general design of parallel branch and bound, specific problems must be implemented for performance experiments. To reach this goal and to complete the parallel design, the next chapter presents details of the parallel N-queens and parallel deadline job scheduling implementations.

V. Parallel N-queens and Parallel Deadline Job Scheduling Implementation

In the previous chapter, a general object-oriented design for the branch and bound search was presented. Using this design, the details of the two search prc 'lems can be addressed. The first class of search, backtracking search, is exemplified by the N-queens problem. The goal of the N-queens search is to place N chess queens on an N-by-N playing board such that no queens are attacking. The characteristics of the N-queens show exponential time complexity along with an exponential number of answer nodes in the solution space. During the backtracking search, the nodes in the solution space are examined using a depth-first while certain branches of the search tree can be pruned using a simple bound function (see Chapter IV, Parallel Branch and Bound). The second class of search, least-cost branch and bound, is modeled by the deadline job scheduling problem. The goal of this problem is to find the largest subset of jobs that can be run by their respective deadlines while minimizing the total penalty incurred. Such a problem can exhibit exponential time complexity in the worst case scenario and linear complexity in the trivial case. In all cases, the deadline job scheduling problem is exemplified by a small number of answer nodes in the solution space. During the least-cost branch and bound search, the nodes in the solution space are examined in least cost order and branches of the search tree are pruned by two bound functions. The discussions of the parallel implementation of these two search problems open with a description of the implicit and explicit constraints as well as the implementation of the solution vector. This is followed by the implementation details of each object as defined in the object-oriented design. The results of the performance experiments are then presented in Chapter VI.

Parallel N-queens Constraints

Before examining the implementation of each branch and bound object, the implicit and explicit constraints as well as the solution vector must be defined (16:337-339). The N-queens problem can be solved with a vector, $(x_1,x_2,\ldots,x_n)$, where x_i is the column on which queen i is placed. For example, the 4-queens problem has a solution vector of (x_1,x_2,x_3,x_4), and the value of x_1 would identify the column number of the playing board in which queen #1 is placed. Each x_i in the vector is bound by explicit and implicit constraints (see Chapter IV, Parallel Branch and Bound, for details). The explicit constraints define those values that can be assigned to each x_i. For the N-queens problem, the explicit constraints are defined as $x_i \in \{1,2,\ldots,N\}$. For example, the explicit constraints for 4-queens problem are $x_i \in \{1,2,3,4\}$. Given the definition of the solution vector and the description of the explicit constraints, a valid solution node in the 4-queens search space would be (1,4,2,3). Figure 20 shows a corresponding diagram of the playing board. Since every node in the solution space meets the explicit constraints, Figure 21 on the next page shows the partial solution space of the 4-queens problem as well. The grey node in the Figure 21 represents the example solution vector (1,4,2,3).

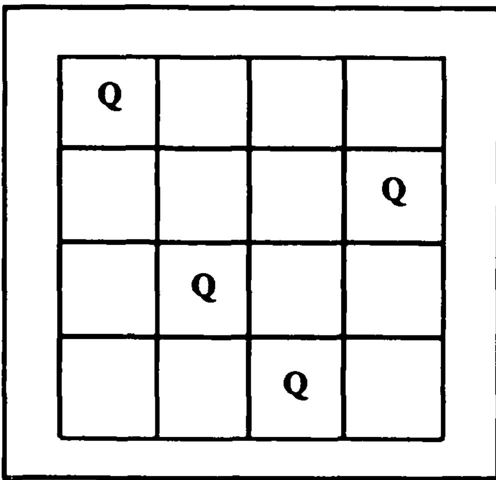


Figure 20: Board positions for solution vector (1,4,2,3)

Figure 21: Partial Solution Space for the 4-queens problem (16:326)

The second set of constraints, implicit constraints, define the relationship among the various x_i's. The nodes in the solution space that meet both the explicit and implicit constraints define answer nodes to the problem. The implicit constraints for the N-queens problem do not allow any queens to be in the same column or to be placed on a common diagonal or subdiagonal. The example solution vector (1,4,2,3) clearly violates these implicit constraints (see Figure 20) and therefore cannot represent an answer node to the 4-queens problem. With these constraints in mind, Figure 22 shows the board positions and the black node in Figure 21 identifies the answer node, (2,4,1,3) to the 4-queens problem.

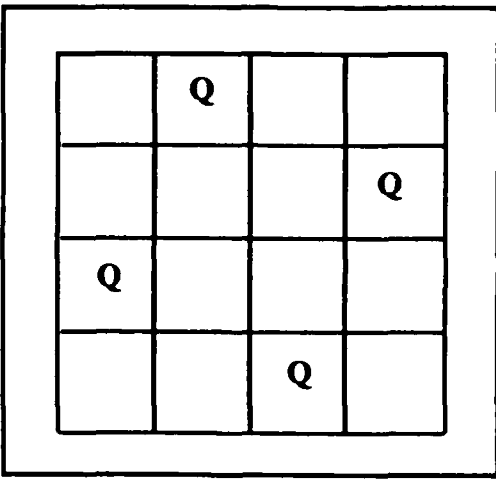


Figure 22: Board positions for answer vector (2,4,1,3)

Based on the implicit and explicit constraints, the implementation of the solution vector for the Parallel N-queens problem is a simple one dimension array of N+1 elements. The first element of the array is used to maintain the state information during the search. Using the first element to maintain the state, an explicit live node queue is no longer needed. All information to conduct the search is contained in the current solution vector. In the case of the N-queens, the state information needed is the identity of the next queen to 'place on the board.' For example, the solution vector (1,3,•,•) shows queen #3 (vector[0]) as the next

queen to place on the board. The remaining elements of the vector correspond to queen #1 placed in column 1 (vector[1]) and queen #2 placed in column 3 (vector[2]). From this example, the solution vector array would have the following values,

$$\textbf{vector[0]} = \textbf{3} \quad \textbf{vector[1]} = \textbf{1} \quad \textbf{vector[2]} = \textbf{3}$$

For the N-queens search, the live nodes are examined in the order they enter the live node queue. Therefore, the branch and bound cost function parameters are defined as follows:

$$\begin{array}{ll} h(X) = 0 & \text{(all nodes have the same cost)} \\ g(X) = 0 & \text{(estimated cost not used)} \\ \hline c'(X) = \text{Depth First Search} & \end{array}$$

The next sections describe the implementation of each object for the parallel N-queens problem. It may be useful to follow the implementation of the parallel N-queens with the visibility diagram developed in the object-oriented design of parallel branch and bound (see Figure 23).

N-queens Control Process

Overview. Focusing on the top processor box in Figure 23, three objects, the meta-controller, the terminate check, and the bound check, work together to control the progress of the parallel search. The meta-controller becomes the central control, the terminate check identifies the stop condition, and the bound check could maintain global bounds throughout the search. Because the N-queens problem does not maintain global constraints, the bound check is not implemented. As recommended by Intel programmers, communications between processes must be minimized for efficient use of the iPSC architecture (17). Therefore, to reduce communications overhead, two objects, the meta-controller and the terminate check are grouped into one iPSC process called the Control Process. A detailed description of both Control Process objects follows.

Processor

Terminate Check

Meta-Controller

Bound Check

. . .

Processor-1

Terminate Check

Problem Solver

Controller

E-Node Expander

Bound Check

Processor-N

Terminate Check

Problem Solver

Controller

E-Node Expander

Bound Check

Figure 23: Object Diagram for Parallel Branch & Bound

Meta-Controller. The meta-controller for the parallel N-queens performs three major operations; (1) keep a list of problems; (2) keep a list of idle worker nodes; and (3) give problems to the worker nodes to solve. First, the list of live nodes (list of problems) is maintained as a FIFO queue data structure. Before the parallel search begins, the meta-controller creates an initial list of problems to solve. This translates to generating the first levels of the search tree. Through experiments, the optimal number of problems to create was determined to be 5 times n, where n is the number of processors in the hypercube. It should be noted the experiments for this thesis have been optimized for large board sizes. Since the N-queens problem is solved using the depth-first search method, all live nodes are expanded in the order they enter the queue. The second task for the meta-controller is to maintain a list of idle worker nodes. Using a status array with two status values, BUSY or AVAILable, the Worker Processes send WORK REQUEST messages to the Meta-Controller when they need work. The meta-controller in turn sets the node status to AVAILable. Finally, the controller must assign problems to the worker nodes. Before a problem can be assigned, two conditions must be met, (1) the live node queue must not be empty, and (2) a worker node must be AVAILable. After passing these two conditions, the meta-controller picks a problem from the live node queue and sends it to the worker. The worker status is then set to BUSY. Notice in the parallel environment, several nodes are expanded simultaneously.

Terminate Check. For the N-queens experiments, the termination condition was to find all solutions. This condition occurs when the live node queue is empty and all of the worker nodes have posted WORK REQUESTS. This translates to a machine state where no more work is available and all of the workers need something to do. Once this condition occurs in the Control Process, the meta-controller sends KILL messages to all Worker Processes.

N-queens Worker Process

Overview. Examining Figure 23 once again, processor #1 through processor #N contain a complete branch and bound problem solver. In this case, for efficiency and reduced communications, all five branch and bound objects are collected to form the iPSC Worker Process. The Worker Process is a modified version of the sequential N-queens code. Most of the modifications include communications functions so the Worker Process can send and receive information from the intermediate host and the Control Process. The details of the Worker Process objects follows.

Controller. The controller object initializes the worker node by receiving the board size from the intermediate host and sending a WORK REQUEST to the Control Process. The controller then enters an infinite loop waiting for a problem to solve. If an E-node message arrives, then hand the E-node to the problem solver. Once the problem solver object finishes, send a WORK REQUEST to the Control Process. If a KILL message arrives, 'break' the infinite loop and terminate the Worker Process.

Problem Solver. The problem solver is contained in a 'while' loop within the controller. Upon receiving an E-node message from the Control Process, the problem solver searches the subtree defined by the E-node. The goal of the problem solver is to find all answer nodes in the subtree.

E-Node Expander. Given an E-node, the next queen to place on the board can be found in the first element of the E-node vector, vector[0]. Using this value, generate all children of the E-node. For example, E-node (2,•,•,•) has the following children, (2,1,•,•) (2,2,•,•) (2,3,•,•) and (2,4,•,•).

Bound Check. The bound check ensures that the implicit constraints of the problem are met. As the children of the E-node are generated, the bound check makes sure that the new vector represents a valid board position. A board position is valid when the new queen is not on the same column or on a common diagonal or subdiagonal to another queen. For example, the E-node (2,•,•,•) has only one valid child, (2,4,•,•). It should be noted if the last queen is placed on the board in a valid position, then this vector defines an answer node.

Terminate Check. The terminate check for the Worker Process is a KILL message from the meta-controller.

The details of the C Language implementation of the Intermediate Host program, the Control Process, and the Worker Process for the parallel N-queens can be found in Appendix A. Besides a description of the source code, a trace of an example problem is shown. For comparison, a sequential version of the N-queens problem was run on a DEC VAX 11/785 and on an Elxsi System 6400. The source code and description of the sequential N-queens can be found in Appendix B. The implementation of the parallel N-queens proved a problem traditionally solved using a sequential computer can be mapped to a parallel architecture. It should be noted that the N-queens search is considered a simple search because no global bounds are maintained and subtrees can be solved independently from the remaining parts of the search space. Therefore, the performance of a more complex search must be examined. The deadline job scheduling problem meets this requirement. Using a least-cost branch and bound technique, the deadline job scheduling problem maintains a global upper bound as well as a global best solution. While these additional constraints are easy to maintain using a sequential processor, they impose additional

complexity to the parallel solution. The following sections describe the deadline job scheduling (DJS) constraints and the implementation of the DJS Control Process and the DJS Worker Process.

Parallel Deadline Job Scheduling Constraints

The ultimate goal of the deadline job scheduling problem is to find the largest subset of jobs that can run by their deadlines while minimizing the total penalty incurred. Before examining the implementation of the parallel search, it is important to understand the constraints and the solution vector of the problem. The deadline job scheduling solution vector, $(x_1,x_2,\ldots,x_n)$, contains a value for each of the n jobs. The explicit constraints for this problem are simply, $x_i \in \{1,0\}$ where 1 denotes that job i is included in the schedule and 0 denotes that job i has not been included into the schedule. For example, a 4-Job problem has a solution vector of (x_1,x_2,x_3,x_4), and a valid solution vector would be $(1,1,1,\bullet)$. This solution vector defines the search state where jobs 1, 2, and 3 have been scheduled and job 4 has not been scheduled. To help the reader in understanding the DJS constraints, the following job set will be used in examples throughout this section (16:384),

Job	p_i	d_i	t_i
1	5	1	1
2	10	3	2
3	6	2	1
4	3	1	1

where,

p_i = the penalty for not scheduling job i.

d_i = the deadline by which job i must be completed.

t_i = the time to run job i.

Using the definition of the explicit constraints, the solution space of the example job set is depicted in Figure 24. The grey node identifies the example solution vector (1,1,1,•). The second set of constraints, implicit constraints, define the relationship among the various x_i's. The nodes in the solution space that meet both the explicit and implicit constraints define answer nodes. The first implicit constraint for the DJS problem is called the Deadline/Total Time Bound. This constraint requires a job to be schedule such that the total run time for all jobs included in the schedule does not exceed the maximum deadline. Referring to the example 4-Job problem above, the solution vector (1,1,•,•) passes the Deadline/Total Time implicit constraint because the maximum deadline of jobs #1 and #2 equals 3 and the total run time of jobs #1 and #2 equals 3. However, the solution vector (1,1,1,•) does not pass the Deadline/Total Time Bound because the maximum deadline of jobs #1, #2, and #3 equals 3 and the total run time of those same jobs equals 4.

The second implicit constraint, Cost/Upper Bound constraint, is based on the cost of the node and a global upper bound. The cost function is calculated in two steps (16:386). First, find **m** where

$$\mathbf{m} = \max\{i \mid i \in S_X\}$$

and S_X = the subset of jobs examined at node X.

Next, compute the cost of node X using the following equation,

$$c'(X) = \sum_{\substack{i \leq m \\ i \notin J}} P_i$$

where J = the set of jobs included in the schedule at node X.

Figure 24: Example 4-Job Deadline Job Scheduling Solution Space

The cost of a node translates to the total penalty paid of all jobs that have not been scheduled so far. The cost of each node of the example job set is shown inside the circles of Figure 24. For example, the cost of solution vector (1,0,•,•) equals 10 because,

$$S_x = \{1,2\}$$

$$m = \max\{i \mid i \in S_x\} = 2$$

$$J = \{1\}$$

$$c'((1,0,\bullet,\bullet)) = \sum_{\substack{i \leq 2 \\ i \in \{2,3,4\}}} p_i = 10$$

The second part of the Cost/Upper Bound constraint is the upper bound of node X. It is defined by the following function,

$$U(x) = \sum_{i \notin J} p_i$$

The value of the upper bound identifies the maximum cost solution node in the subtree rooted at node X. For example, node (0,1,•,•) of the tree in Figure 24 has an upper bound of 14 since the cost of solution node (0,1,0,0) equals 14 and solution node (0,1,0,0) is the highest cost node in the subtree. During the search the lowest upper bound is kept as a global bound. The global upper bound is defined by the following function,

global upper bound = min{U(x), current global upper bound}

where, x is a child of the current E-node.

Based on the cost of the node and the present value of the global upper bound, a child of an E-node becomes a live node only if the cost of the child is less than the global upper bound.

From the definition of the solution vector and the DJS constraints, the implementation of the solution vector for the parallel job scheduling problem is represented as a C Language structure (Pascal and Ada record type). The first field of the structure is the state vector defined by a one dimensional array of N + 1 elements. Following the model in the N-queens problem, the first element of the array is used to maintain the state of the search. Hence, the value in vector[0] identifies the 'next job to schedule.' For example, the solution vector (1,0,•,•) identifies the problem state where job #3 (vector[0]) is the next job to schedule, job #1 is included in the schedule (vector[1]) and job #2 is not included in the schedule (vector[2]). From this example, the state vector would have the following values,

vector[0] = 3   vector[1] = 1   vector[2] = 0

The second field in the C Language structure is the cost of the state vector. Because the state vector represents a node in the search space, the cost of the state vector can be determined using the method described in the Cost/Upper Bound discussion.

For the deadline job scheduling search, the live nodes are examined in least-cost order. Therefore, the branch and bound cost function parameters are defined as follows:

h(X) = Cost of node X
g(X) = Upper Bound

---

c'(X) = Least-Cost Branch and Bound

The next sections describe the implementation of each object for the parallel deadline job scheduling problem. It may be useful to follow the implementation with the visibility diagram developed in the object-oriented design of parallel branch and bound (see Figure 25).

Deadline Job Scheduling Control Process

Overview. Focusing on the top processor box in Figure 25, three objects, the meta-controller, the terminate check, and the bound check, work together to control the progress of the parallel search. The meta-controller becomes the central control, the terminate check identifies the stop condition, and the bound check maintains global bound throughout the search. As recommended by Intel programmers, communications between iPSC processes must be minimized for efficient use of this parallel architecture (17). Therefore, to reduce communications overhead, three objects, the meta-controller, the bound check, and the terminate check are grouped into one iPSC process called the Control Process. A detailed description of the Control Process objects follows.

Meta-Controller. The meta-controller for the parallel deadline job scheduling performs five operations;

(1) Keep a list of problems
(2) Keep a list of idle worker nodes
(3) Give problems to the worker nodes to solve
(4) Maintain the global upper bound
(5) Maintain the global best solution

First, the list of problems (live nodes) are kept in a priority queue data structure. This queue maintains the live nodes in least cost to highest cost order. From this data structure, the least-cost live node is always expanded first and hence the name, least-cost branch and bound. Second, the controller maintains the list of idle worker nodes using a status array with two status values, BUSY or AVAILable. The workers send WORK REQUEST

Figure 25: Object Diagram for Parallel Branch & Bound

messages to the controller when they need work. The controller in turn sets the node status to AVAILable. Third, the controller assigns problems to worker nodes. Before a problem can be assigned, two conditions must be met, (1) the live node queue must not be empty, and (2) a worker node must be AVAILable. After passing these two conditions, the controller picks the live node from the front of the queue and sends it to the worker. The worker status is then set to BUSY. Finally, the meta-controller collects intermediate answers from the Worker Processes. These 'local' best answers are handed to the bound check where the global upper bound and the global best answer is maintained.

Terminate Check. The terminate condition for deadline job scheduling problem is to find the first solution. This condition occurs when the live node queue is empty and all of the worker nodes have posted WORK REQUESTS. This translates to a machine state where no more work is available and all of the workers need something to do. Once this state occurs, the Control Process sends a KILL message to all Worker Processes.

Bound Check. Workers calculate 'local' best solutions and send them to the meta-controller. As the local 'best' nodes enter, the bound check compares the upper bound of this new 'best' node to the current global upper bound. If the upper bound of the new live node is less than the present upper bound, then a new best solution has been found and a new upper bound has been calculated. The meta-controller stores this node as the best solution and broadcasts new upper bound to all workers.

Deadline Job Scheduling Worker Process

Overview. Examining Figure 25 once again, processor #1 through processor #N contain a complete branch and bound problem solver. In this case, for efficiency and reduced communications, all five objects are collected to form the iPSC Worker Process. Because of the lack of memory at an individual iPSC node, the DJS Worker Process solves the job scheduling problem using a different method as compared to its sequential counterpart. While the sequential version of deadline job scheduling conducts a complete least-cost branch and bound search, the iPSC Worker Process uses a 'blind' depth-first search of the solution space. The depth-first method can be conducted without an explicit queue and therefore is not impaired by a memory space limit. The following sections describe the iPSC DJS Worker objects.

Controller. The controller initializes the worker node by receiving the list of jobs to schedule from the intermediate host and by posting a WORK REQUEST to the Control Process. The controller then enters an infinite loop waiting for a problem to solve. If an E-node message arrives, then send the E-node to the problem solver object. Once the problem solver finishes, send a WORK REQUEST to the Control Process. The controller terminates upon receiving a KILL message from the Control Process.

Problem Solver. The problem solver receives an E-node (a subtree in the search space) from the control object. The goal of the problem solver is to search the subtree rooted at the E-node. To do this, the problem solver conducts a simple depth-first search of the subtree. During the search, the problem solver receives the most recent global upper bound from the meta-controller. Additionally, the problem solver keeps track of the local 'best' solution. Upon finishing the search of the subtree, the problem solver sends the new best solution to

the Control Process. It should be noted the new 'best' node is only a local optimum and it is the responsibility of the Control Process to keep track of the global best answer.

E-Node Expander. An E-node is expanded by calculating two children, (1) child one attempts to schedule the next job; and (2) child 2 does not schedule the next job. For example, E-node (1,1,•,•) of a 4-Job problem has the following children, (1,1,1,•), and (1,1,0,•).

Bound Check. As the children of the E-node are generated, the bound checks insure that the next job can be scheduled by evaluating the Deadline/Total Time Bound and the Cost/Upper Bound constraints. Using the 4-Job example, the first child of E-node (1,1,•,•) is (1,1,1,•). This child does not pass the Deadline/ Total Time Bound because jobs #1, #2, and #3 have a total run time of 4 and a maximum deadline of 3. Hence, child 1 does not become the next live node. The second child, (1,1,0,•), may or may not be valid because of the dependency upon the value of the global upper bound when E-node (1,1,•,•) is expanded. The cost of node (1,1,0,•) is 6. It will become a live node if the global upper bound is greater than 6.

Terminate Check. The Worker Process terminates when the Control Process sends a KILL message.

The details of the C Language implementation of the Intermediate Host program, the Control Process, and the Worker Process for the parallel deadline job scheduling can be found in Appendix C. For comparison, a sequential version of the job scheduling problem was run on a DEC VAX 11/785. The source code description of sequential DJS can be

found in Appendix D. A trace of an example problem is also included in Appendix D. The implementation of the parallel DJS proves once again a problem traditionally solved using a sequential computer can be mapped to a parallel architecture. Even though the Worker processes in the parallel use a different search method, this implementation shows how a more complex search with global bounds and a global best answer can be solved concurrently.

Conclusions

The implementations of the parallel N-queens and parallel deadline job scheduling follow the object-oriented design developed in Chapter IV. While the details of the actual source code was not presented, the implementation of the objects and the packaging into iPSC processes was defined. Using these implementations of parallel search, the performance of these two problem can now be measured. The next two chapters defines the performance measures used in this research and presents an analysis of the experimental results.

VI. Performance Analysis and Experimental Results

The implementation details of the two search problems, N-queens and deadline job scheduling were presented in the last chapter. Once these implementations were tested and validated, a series of performance experiments were run. Many times one experiment needed additional tests to explain and justify the results. Hence, three measures, Computation Time, Speed Up, and Load Balance, are used to categorize the performance of the parallel search problems and the iPSC computer. These measures were selected for three reasons, (1) most researchers present these results in the literature; (2) most of the more complex performance measures are beyond the scope of this research; and (3) these measures are valid for MIMD programmed machines. (It seems most parallel performance research is constrained to SIMD evaluations. See (27) for details). In the next sections, the three performance measures, the sequential baseline as well as the results of the parallel N-queens and the parallel deadline job scheduling are described.

Computation Time

The first measure of an algorithm's performance is run time. Because the parallel computational environment involves additional processes, one representation of the total computation time of an algorithm is defined by the following formula (1:95),

$$T_N = T_s + T_c + T_w \qquad (1)$$

where

T_N = Computation Time for N processors
T_s = Start Up Time
T_c = Processors Computation Time
T_w = Wind Down Time

Start Up time, T_s, measures the time to initialize the parallel processor before any parallel computations can begin. Start up may include such things as initial parsing of the job, initial message transfers, or down load time of the programs to the parallel processor itself. The second term, processor computation time T_c, measures the time the computer spends actually solving the problem. This term is common in sequential processor run time analysis. The final term in equation 1 is wind down time, T_w. This time accounts for the gathering of results from the various processors in the computer and analyzing or tallying those final results.

Speed-Up

The second performance measure, speed-up, compares the time to compute a solution using one processor and the time to compute a solution using N processors. It is defined as follows (26:28),

$$S = T_1 / T_N \quad (2)$$

where

T_1 = Time to compute a result with one processor
T_N = Time to compute a result with N processors (Eqn 1)

The speed up of a problem run in parallel is easy to understand. If a problem can be parsed into N subproblems, with each subproblem taking 1/N in the total computation, then the maximum speed-up of N is achieved. The perfect speed up, N, is highly unlikely because the overhead of the start up and the wind down time as well as the communications among processing elements induce limitations on this measure. Properly, the T_1 time should be a d-0 hypercube. Unfortunately, a single iPSC node could not support a queue large enough to solve large deadline job scheduling problems. Therefore, the T_1 time is calculated using VAX 11/785. The T_N times are calculated for various size iPSC cubes.

Load Balance

The third performance measure is load balance. Because of the nature of the design and the branch and bound problem, the load is defined to be the number of E-nodes examined by a Worker Process. Even though this is a simple measure, when plotted against the average load performed across all worker processors balanced and unbalanced work loads can be identified. Single Instruction Multiple Data (SIMD) problems that partition data to promote parallel activity tend to have regular communication and computation cycles. These problems show the best performance under balanced work loads (19). Since parallel search is a Multiple Instruction Multiple Data (MIMD) problem, the communication and computation cycles cannot be guaranteed to be regular. Hence, the load balance measure must be evaluated along with the other performance measures before conclusions can be reached.

Baseline Performance

The sequential baseline for this research is a Digital Equipment Corporation VAX 11/785 running the 4.2 BSD (Berkeley Software Distribution) UNIX operating system. The VAX is used as the sequential baseline for two reasons, (1) a VAX was available for this research; and (2) the VAX is the defacto industry standard for performance measures. The configuration of the machine used for this research has 8 Megabytes of main memory and 1800 Megabytes of disk storage. The sequential versions of the N-queens and deadline job scheduling problems were programmed in C Language. The source code for both problems is in Appendices B and D respectively. The time information was obtained using the UNIX "times" function. Of the four parameters measured by the "times" function, this research focused on user_time. The user_time of a process is that time devoted to computation. The overhead associated with system calls, page swaps, etc. was not used for two reasons, (1)

this research is actually interested in compute time of the algorithm and not operating system overhead; and (2) the VAX is under various system loads during the course of the experiments which would influence system time and the overall timing data.

Because the "times" function has a resolution of 1/60th of a second, the following procedure was used to calculate the benchmark times. First, each of the N-queens and deadline job scheduling problems was run ten times. Then, the highest time answer and the lowest time answer were thrown away and the remaining eight times were averaged. Several experimental runs show that the computation time calculated by the "times" function varied by one or two clock ticks. Therefore, this averaging procedure reflects an accurate timing analysis of the problem. It is also important to note for small problem sizes the 1/60th of a second resolution resulted in some 'unmeasurable' test runs.

An Elxsi System 6400 supercomputer was also available for this research. The sequential version of the N-queens that was run on the VAX was also run on the Elxsi. The Elxsi was unavailable for sequential deadline job scheduling tests. This machine is configured with 16 Megabytes of main memory and 1896 Megabytes of disk storage. The timing results were compiled using the UNIX "times" function and the same averaging procedure as on the VAX.

Parallel Performance Experiments

The results of the parallel versions of the N-queens and deadline job scheduling (DJS) are presented in this section. Before listing the results, the experimental procedure should be identified. After the parallel design, each problem was tested on the Intel iPSC simulator running on a VAX. While the simulator creates a good environment to learn how to program

an iPSC, it does not show true parallel activity. Therefore, it should not be used to fine tune a problem. After porting the code to the iPSC, the original design was modified to achieve the best computation times. It should be noted object design worked well for an initial implementation, but the best performance results were attributed to fine tuning on the actual hardware. Details on the actions taken to fine tune the N-queens and deadline job scheduling are discussed below. Once the run time versions of the problems were coded, tested, and validated, then each problem was run on several cube sizes. The N-queens problems were run on d-0, d-1, d-2, d-3, d-4, and d-5 hypercubes (where d=dimension). The deadline job scheduling problems were run on d-1, d-2, d-3, d-4, and d-5 hypercubes. The d-0 cube did not have enough memory for the DJS d-0 experiments. For each experiment, an averaging procedure is used to calculate the timing results. The averaging steps included 10 separate runs, removing the highest and the lowest times, and then averaging the remaining eight results. The timing function for all runs was calculated using the iPSC Clock function on the nodes. The resolution of the iPSC Node Clock function is 1/60th of a second. Once again, the timing experiments show that the iPSC computation time for a particular problem was within a 2 or 3 clock tick resolution. Therefore, the averaging procedure represents an accurate measure of the computation time.

Parallel N-queens. Tables of the performance measures for the parallel N-queens experiments can be found in Appendix E. The data shown in this chapter has been plotted to show trends and for comparisons. Upon porting the parallel N-queens software from the iPSC Simulator to the actual iPSC, the original design was modified to achieve the best computation times. The only parameter of the parallel N-queens used for fine tuning resides in the iPSC Control Process. This process has the responsibility to create the initial set of problems to solve. At some point in time, it becomes beneficial to stop creating problems

and to start handing them out to worker nodes. The results of the experiments for this analysis have been tuned to large problem sizes by creating 5 time *n* problems, where *n* equals the number of processors in the cube.

First, an analysis of the time to compute the first solution of the N-queens problems represents the simple case of an answer to a search problem. If any answer to a search meets the requirements of the problem, then the time to first solution is important. Since the iPSC has the additional overhead of a start up time, wind down time, as well as communications among the processors, the hypercube matched the VAX and Elxsi while solving large problems (see Figure 26). Therefore, for small problems the overhead appears to be excessive. However, for larger problems (13-queens or greater), the advantage of parallel processing becomes apparent.

Second, an analysis of the time to compute all solutions of the N-queens problem represents the opposite case of answers to a search problem. These experiments model the search for an optimal solution. As the computation time and speed up measures show, the suitability of parallel processing is dependent on the problem size. Once again, for small problems, the overhead of start up time and communications time within the iPSC overwhelms the computation time of the problem. On the other hand, for large problems, parallel processing demonstrates the possibilities for increased performance. First, the inherent exponential behavior of search problems is shown in Figure 27. Cognizant of the semi-log plot and concentrating on the VAX and Elxsi curves, the time to all solutions shows true exponential time complexity. Notice that the iPSC d-4 and d-5 curves cross the VAX at approximately the 8-queens mark. This translates to a speed up of one (see Figure 28). Continuing the analysis, the d-4 and d-5 hypercubes show significant speed ups between the

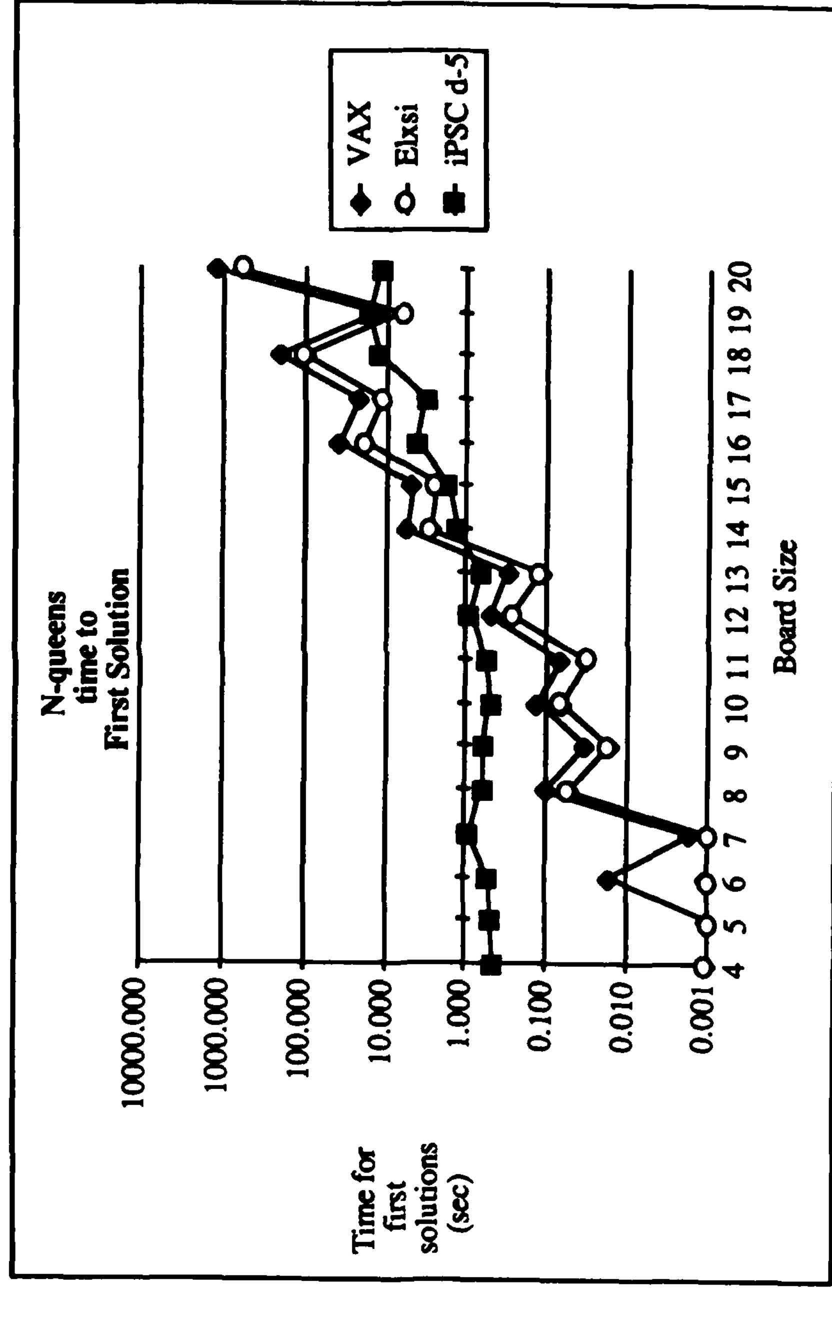


Figure 26: N-queens Time to First Solution

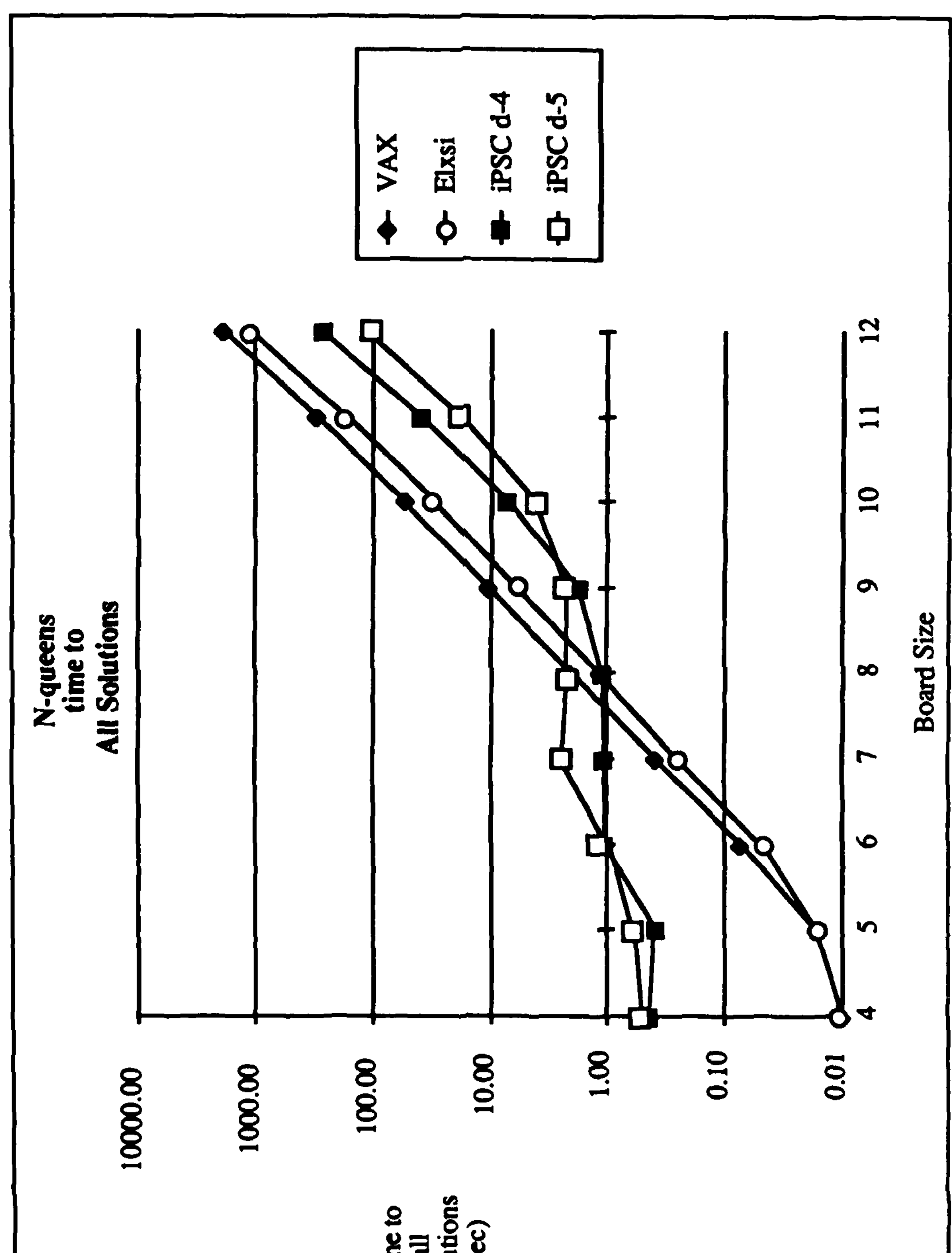


Figure 27: N-queens Time to All solutions

board sizes of 8 and 10. Beyond the 10-queens problem, the inherent exponential nature of the search is introduced once again. For problems larger than 10-queens, the iPSC shows 'maximum parallelism' with a constant speed up. The d-4 levels to a constant speed up of ~7 over VAX and the d-5 hypercube reaches a steady state speed up of ~16 over VAX. The speed up curves in Figure 28 reinforce this analysis by showing the reduction to the time order complexity (increasing speed ups) followed by a leveling to an approximate constant. It should be noted that smaller cube sizes show similar exponential complexity trends. (The plots for the small cube runs are not shown, refer to the tables in Appendix E). While the d-3 hypercube reaches a constant speed up of ~5, the d-2 cube and the Elxsi have approximately the same speed up of 1.7 over VAX. Small hypercubes, 1-d and 0-d, show no advantage to parallel processing with speed ups of 0.5 and 0.1 respectively.

One final observation can be made about the time to all solutions. Figure 27 shows that the d-4 hypercube actually computed all solutions faster than a d-5 hypercube for small job sizes. This phenomena is attributed to the fine tuning parameter as described in the beginning of this section. For these experiments, the Control Process created 5 times n problems to solve, where n is the number of nodes in the cube. This value was selected to get the best performance from large problem sizes. If the Control Process creates 2 times the number of nodes in the cube of initial problems, then the system is tuned to the 8, 9, and 10-queens problem sizes (see Figure 29). Therefore, the parallel search can be tuned to run efficiently if the range of problem sizes are known.

Figure 28: iPSC and Elxsi Speed Up Over VAX

N-queens Time to All Solutions
Optimized for Board Sizes of 8, 9, & 10

Time (sec)

12
10
8
6
4
2
0

4 5 6 7 8 9 10

Board Size

iPSC d-4
iPSC d-5

Figure 29: Time to All solutions Optimized for Board Sizes of 8, 9, & 10

Finally, the load balance analysis shows that even though the N-queens is an MIMD problem that exhibits irregular computation and communications cycles, the work load was balanced across all worker nodes. In Figure 30, the load balance for the 11-queens problem running on a d-5 hypercube is shown. This plot is typical for all board sizes of 8 or greater. The line denotes the average work load across all nodes and the dots represent the actual work load at a particular node. The best speed ups for problems occurred using a large dimension cube with large problem sizes. Since the range of best performance coincides with the range of equivalent load balance, an even load balance may be desirable while designing a parallel solution to 'backtracking' search problems.

Load Balance for the
11-queens running on a D-5 cube

Enodes
Expanded

12000
10000
8000
6000
4000
2000
0

1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31

Node Number

Figure 30: Load Balance for the 11-queens on a D-5 Hypercube

AD-A179 384 PERFORMANCE EVALUATION OF PARALLEL BRANCH AND BOUND SEARCH WITH THE INTEL..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. R T MRAZ 2/3

UNCLASSIFIED DEC 86 AFIT/GCE/ENG/86D-2 F/G 9/2 NL

MI

In summary, the results of the N-queens tests show parallel processing can reduce the time order complexity of a simple 'backtracking' search over a narrow range of problem sizes. At some point after this reduction in the time complexity, the inherent exponential nature of the search problem is introduced once again. These results also show that parallel processing may show speed ups while solving a simple search (first answer) to more difficult searches (optimal answer). With these results in hand, an examination of a more complex search is in order.

Parallel Deadline Job Scheduling. Tables of the performance measures for the parallel DJS experiments can be found in Appendix E. The data shown in this chapter has been plotted to show trends and for comparisons. Before analyzing the results of the job scheduling experiments, a description of the test data is necessary. Since the deadline job scheduling solution uses least-cost branch & bound, the time to schedule a set of **n+1** jobs may take less time than scheduling **n** jobs. Therefore, two pseudo-equivalent classes of problems were devised such that the more jobs to scheduled create a more difficult problem to solve. Two reasons for creating pseudo-equivalent classes are, (1) the proof of equivalent classes of jobs is beyond the scope of this research; and (2) job sets with these characteristics make the analysis a bit easier.

With this background, the first set of problems guarantees that all jobs can be scheduled. The VAX solves this set of problems in O(n) time. As described in Chapter V, each job is defined by a 3-tuple (p_i, d_i, t_i), where p_i is the penalty paid if the job is not scheduled, d_i is the deadline when the job must be finished running, and t_i is the time to run job i. Creating a set of **n** jobs with the following characteristics guarantees that all jobs can be scheduled,

$$p_i = 1, \forall \ i$$

$$\sum_{i=1}^{n} t_i \leq \min(d_i)$$

The following list shows a typical job mix for problem set #1,

Job	t_i	p_i	d_i
1	2	1	100
2	3	1	200
3	2	1	300
4	3	1	400

The second pseudo-equivalent class is described with the following values for p_i, d_i, and t_i,

$$t_i = i$$

$$p_i = 2*t_i$$

$$d_i = \left\lfloor \frac{\sum_{i=1}^{n} t_i}{2} \right\rfloor$$

The VAX solves problems defined with these parameters in exponential time. The following table shows a typical job list for the second problem set,

Job	t_i	p_i	d_i
1	1	2	5
2	2	4	5
3	3	6	5
4	4	8	5

Upon porting the parallel DJS software from the iPSC Simulator to the actual iPSC, the original design was modified to achieve the best computation times. The only parameter of the parallel DJS used for fine tuning resides in the iPSC Control Process. This process has the responsibility to create the initial set of problems to solve. At some point in time, it becomes beneficial to stop creating problems and to start handing them out to worker nodes. For the first set of DJS problems, n problems are created, where n equals the number of processors in the cube. For the second set of DJS problems, 4 times n problems are generated. It should be noted that these parameters were selected to get the best performance from large problem sizes.

First, an analysis of the first job set. Parallel processing appears to show no reductions in the time order complexity of O(n) problems (Figure 31). The best performance was attributed to the iPSC d-1 and the best speed up was approximately 0.33 over VAX. Since this search problem degenerates to an examination of the left-most branch of the search tree, the problem does not map well to a parallel processor. The Load Balance analysis shows this result (Figure 32). Basically, this problem cannot be run in parallel. For small problem sizes (scheduling 15 jobs or less) only one processor solves the problem and for large problem sizes two processors are used. This problem re-enforces the concept of maximum parallel activity because of limitations inherent to the problem.

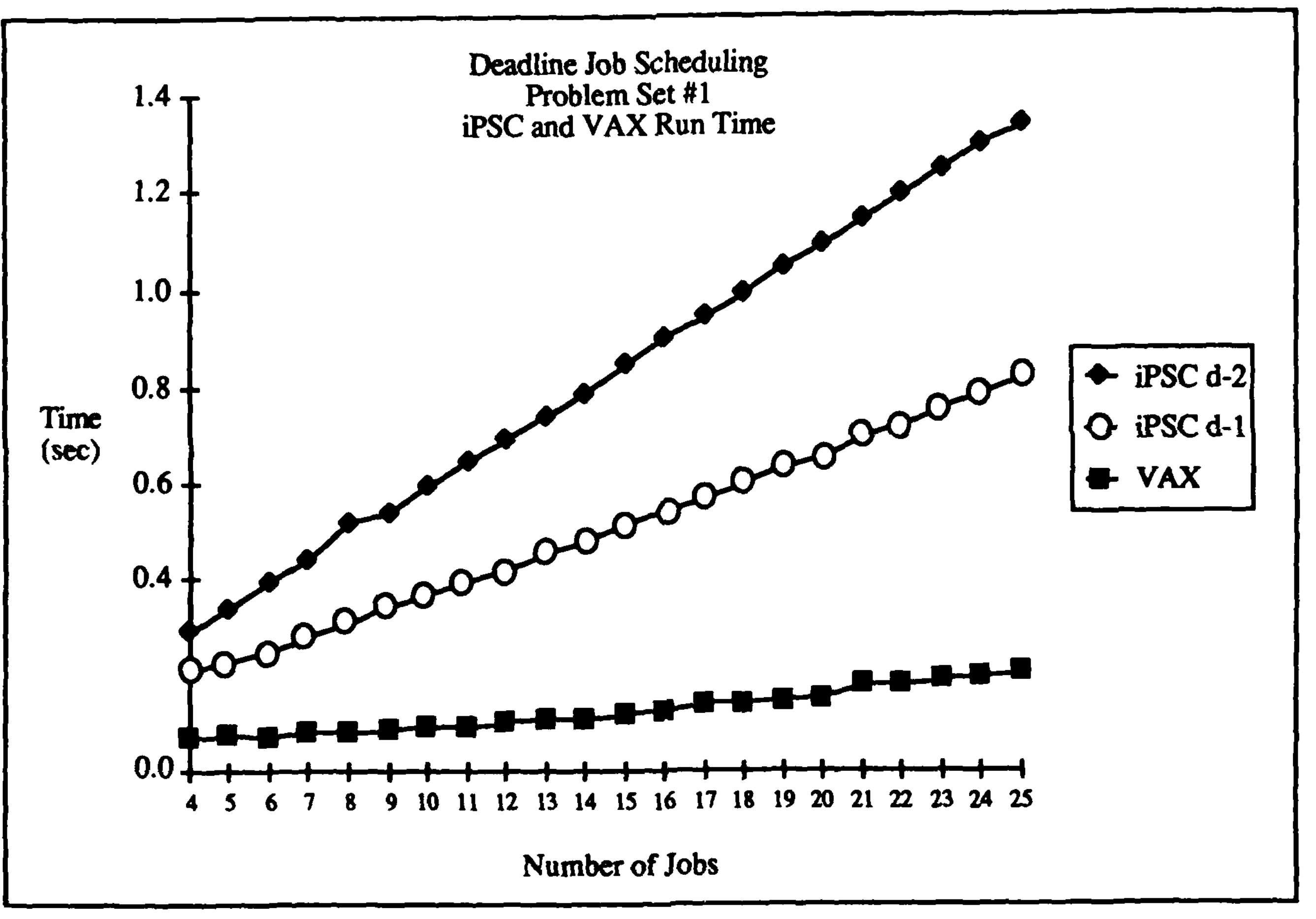


Figure 31: Deadline Job Scheduling- Problem Set #1
Computation Time

Figure 32: Deadline Job Scheduling- Problem Set #1
Load Balance of scheudling 20-Jobs on an iPSC D-4

Next, the exponential class of job schedules must be examined. The run time analysis shows that significant reductions in the time order complexity can be achieved. In Figure 33, the semi-log plot of the VAX computation time shows the inherent exponential nature of the problem. The iPSC d-4 and d-5 curves demonstrate the power of the parallel computation with problem sizes of 11 or greater. It should be noted the the best speed up achieved was 58 times over VAX with a d-5 hypercube solving a 15-Job problem. The d-4 reached a speed up of 43 times over VAX (see Figure 34). A dimension-3 cube achieved a speed up 33 times over VAX. Finally, d-2 and d-1 hypercubes solved the problems approximately 3 times faster than the VAX. These results can be explained while examining the global upper bound during the parallel computation. Figure 35 shows the load balance of scheduling 15 jobs on a d-4 hypercube. Even though the load is unbalanced, the iPSC solved this problem 43 times faster than the VAX. Solving this same problem on a d-5 cube, all worker nodes have E-node Expansion Counts (loads) equal to zero. This anomaly is attributed to the global upper bound. In a d-4 cube, the Control Process generates 64 initial problems. As the workers solve these problems concurrently, the global upper bound converges quickly to the best upper bound in the entire search space. Once the upper bound converges, the workers no longer search the subtrees. They only prune the remaining search space. In the case of the d-5 hypercube, the Control Process generates 128 initial problems to solve. At this point the upper bound has already converged, and the search quickly ends with the workers only pruning the search space and never actually searching a subtree.

Deadline Job Scheduling
Problem Set #2
iPSC and VAX Run Time

Time to first solution (sec)

1000.00
100.00
10.00
1.00
0.10
0.01

4 5 6 7 8 9 10 11 12 13 14 15

Number of Jobs

VAX
iPSC d-5
iPSC d-4

Figure 33: Deadline Job Scheduling- Problem Set #2
Computation Time

Figure 34: Deadline Job Scheduling- Problem Set #2 Speed Up Over VAX

Load Balance - Deadline Job Scheduling
Problem Set #2
15-Jobs solved on an iPSC D-4

E-nodes Expanded

Node Number

Figure 35: Deadline Job Scheduling- Problem Set #2
Load Balance of scheduling 15-Jobs on an iPSC D-4

In summary, the results of the deadline job scheduling problems show that a more complex search with global constraints can be mapped to a parallel architecture. The experiments with the O(n) job set re-enforces the concept of maximum parallel activity due to limitations within the problem. On the other hand, the tests of the exponential job set show that significant reductions in the time order complexity can be achieved for complex search techniques.

Conclusions

The results from running parallel search experiments on the Intel iPSC justify the fundamental issues of this research. Search techniques can be mapped to parallel architectures and speed ups can be achieved. The results also emphasize the strong dependency among the problem, the size of the problem, the parallel design, and the parallel computer architecture. With these results in hand, the next chapter completes this research with final conclusions and recommendations.

VII. Conclusions and Recommendations

The performance evaluation of parallel branch and bound search is almost complete. The analysis, design, and experiments are finished, but the conclusions to this research must be summarized. First, a review of the research is appropriate. In Chapters I and II the fundamental issues, the classes of problems, and the parallel environment for this thesis was established. Next, in Chapter III, an analysis of parallel design was conducted to investigate the process of mapping a problem into a parallel computer architecture. The results of this investigation selected the object-oriented design methodology. Chapter III continued with a discussion of the object model as well as a presentation of a formal object design approach. Using the object model, Chapter IV developed a general parallel branch and bound design. In preparation for the performance experiments, Chapter V described the implementation details of the parallel N-queens and parallel deadline job scheduling problems. Chapter VI concluded the performance tests with descriptions of the measures and an analysis of the experimental results. To finish the research, this chapter presents the final conclusions and recommendations in four parts, (1) Parallel Design Methodology, (2) Performance of Parallel Branch and Bound, (3) Suitability of the hypercube architecture for parallel search, and (4) Recommendations.

Parallel Design Methodology

The results of the first goal identified the object-oriented design methodology as a good design approach to map a problem into a parallel solution. The object model worked well for this research. Object design created a fine grained mapping of the problem space, and the implementation of the design focused on collecting several objects into coarse grained iPSC processes. During the design, details of the branch and bound problem were not overlooked

and during the implementation, inefficiencies of communications were reduced. Even though the initial design needed fine tuning to achieve the best performance, the implementation of the initial design created a good prototype. One reason for the success of the object design methodology is similarity between the object model and the iPSC process model of computation. Therefore, this research recommends the object-oriented design methodology as a parallel design strategy for the hypercube architecture.

Performance of Parallel Branch and Bound

To meet the second goal of this research, the performance of the class of search problems was measured on a parallel processor. As the results show, a sequential problem solving technique, like search, can be mapped to a parallel processor and speed ups over traditional sequential machines can be achieved. In fact, over a narrow range, the parallel solution reduced the time order complexity of the problem. The results of these experiments also show speed ups while computing answers to simple 'backtracking' search as well as more complex search problems that maintain global constraints.

Overall, the research concludes the performance of parallel branch and bound is dependent on the size of the problem, the parallel design of the problem, and the parallel computer architecture. In the case of the size of the problem, if any answer or if an optimal answer to a search problem is needed, then parallel processing may reduce the time complexity for large problem sizes. In the case of parallel design, the results show the hypercube architecture reduces the time order complexity over a range of problem sizes until some point where the exponential nature of search is introduced once again. These results re-enforce to concept of maximum parallel activity because of limitations inherent in the parallel design and inherent to the problem. Finally, in the case of the parallel computer

architecture, the performance of search was improved using a parallel processor. This performance needs additional analysis. Search is a sequential control strategy that was mapped to a loosely coupled parallel architecture. Because search is a tightly coupled technique and because the hypercube is a loosely coupled architecture, some performance degradation is possible.

Suitability of the iPSC for Search Problems

The third goal of this research was to examine the suitability of the hypercube architecture to solve search problems. While the results of these experiments may not meet the 'real-time' requirements of the SDIO and Pilot's Associate researchers, this research recommends the hypercube architecture to solve search problems in parallel. An additional analysis justifies this conclusion. First, the Intel iPSC shows reductions in the time complexity and speed ups of 58 times over VAX. Since one iPSC node has far less computational power as compared to a single VAX 11/785, one can envision much more powerful processors or custom processors configured in a hypercube topology. Such a machine could possibly have the capacity to meet the 'real-time' needs for these researchers.

Recommendations

Several topics for continued research in parallel processing can be recommended. First, the object-oriented design methodology must be evaluated as a general purpose parallel design strategy. The object model worked well for this research because of its similarity with the iPSC process model of computation. Before object design can be used as a general purpose parallel design strategy, it must be exercised by mapping other problems to the hypercube architecture as well as to other parallel architectures. Second, the performance of the class of search problems must be measured on other parallel architectures. It should be

noted that branch and bound is a sequential programming technique with central control. This tightly coupled algorithm was mapped onto an extremely loosely coupled computer hypercube architecture. Even though this research successfully produced speed ups, the nature of the hypercube and the nature of the problem are not similar. Therefore, experiments with parallel search on more tightly coupled architectures, such as shared memory machines, should be conducted. Third, research in new algorithms to solve search problems should be examined. This research mapped a sequential programming technique to a parallel processor. In this case, search shows an inherent exponential time complexity. Research in new algorithms along with the distributed nature of a parallel computer may be able to reduce the time order complexity over a larger range of problem sizes or over the entire range of problems. Finally, the hypercube architecture must be evaluated as a solution to other classes of problems. The hypercube architecture along with its general process model of computation create a good environment to conduct research for a several classes of problems.

In closing, the contribution of this research must be explained. Parallel processing is still in its infancy. Several parallel computers are commercially available but the classes of problems best suited for these machines, parallel design methodologies, and software development environments have been slow to develop. Despite the state of knowledge about such a complex topics, speculation about the benefits of parallel processing is quite abundant. This speculation is usually not supported by analytic or experimental justifications. This research has reached conclusions about the benefits of parallel processing and parallel search in particular. These conclusions have been supported by mapping a parallel search algorithm to the Intel iPSC hypercube computer. Hence the results of this thesis can be used as a benchmark for continued study in parallel search algorithms.

Appendix A:

iPSC N-queens Source Code Description

This appendix describes the source code for the parallel version of the N-queens problem. The appendix is divided into four sections, (1) intermediate host code, (2) control process code, (3) worker process code, and (4) an example trace of the 4-queens problem. The source code for the intermediate host program is shown on the next five pages. This program is the link between the user and the parallel N-queens software that runs on the iPSC. First, the host queries the user for the board size of the problem, the dimension of the hypercube, and the number of times the problem should be run. Next, the host initializes the computation time variables. Then, the host down loads the "control" program and the "worker" program to the iPSC. This is followed by sending all nodes initialization information, the board size of the problem. The control process also needs to know the number of nodes in the cube. After these transactions, the host waits for the 'problem complete' message from the control process. Upon receiving that message, the host collects the timing data. Once all repetitions have been run, the average time to first solution and the average time to find all solutions is reported.

```
/*********************************************************************
*                              THESIS                                *
*                                                                    *
*                         N-queens Problem                           *
*                                                                    *
*                   Intermediate Host Program                        *
*                                                                    *
*  Date: 24 Oct 1986                                                 *
*                                                                    *
*  Functions:  This is the iPSC intermediate host program for        *
*              the N-queens Problem.  This program loads the         *
*              iPSC with the Worker and Control Processes,           *
*              and then waits for an answer message from the         *
*              Control Process.  Upon receiving this message,        *
*              the Host Program retrieves the Timing Data to         *
*              calculate the Time to First Solution and the          *
*              Time to All Solutions of the N-queens.                *
*                                                                    *
*              The averaging method to compute the time              *
*              information is quite simple.  First, run the          *
*              N-queens problem for 'n'  repetitions (a user         *
*              entered parameter).  Next, remove the maximum         *
*              time as well as the minimum time and average          *
*              the remaining numbers.  It should be noted that       *
*              the minimum number of repetitions that this           *
*              program will run is 3.                                *
*                                                                    *
*              The user must enter three parameters for this         *
*              program to run, (1) Board Size, (2) Cube              *
*              Dimension, and (3) Number of Repetitions.  Make       *
*              sure the iPSC is initialized to the same              *
*              dimension as the input parameter.                     *
*                                                                    *
*  Libraries:  Standard Input & Outpu                                *
*              iPSC chost definitions                                *
*                                                                    *
*  iPSC Operating System: Release 2.1                                *
*                                                                    *
*  Language:   C Language                                            *
*                                                                    *
*  Input Parameters: Board Size of the Problem                       *
*                    Dimension of the iPSC                           *
*                    Number of Repetitions                           *
*                                                                    *
*  Outputs: Time to First Solution                                   *
*           Time to All Solutions                                    *
*                                                                    *
*  Author: Capt Rick Mraz                                            *
*                                                                    *
*********************************************************************/
```

```
/********************************************
*              HEADER FILES                 *
********************************************/
#include <stdio.h>
#include "/usr/ipsc/lib/chost.def"


/********************************************
*            Define Constants               *
********************************************/
#define NODE_PID                    0        /* Node Process ID            */
#define HOST_PID                    1        /* Host Process ID            */
#define BOARD_SIZE                  10       /* Send Board Size to nodes */
#define TIMES                       100      /* Get Time data from nodes */
#define NUM_NODES                   130      /* Send number of nodes to
                                                the Control Process     */
#define INFINITY                    9000000  /* Need a big number ???    */
#define VECTOR_SIZE                 12       /* Maximum Board Size       */


/********************************************
 *      Define Functions and Globals        *
********************************************/
long min();                                  /* Use this to get the time
                                                to the first solution   */
int  number_of_nodes;                        /* Number of nodes in cube  */


/****************************************************
*                Main Program                       *
****************************************************/
main()
{
int  board_size,                             /* N-queens Board Size    */
     cid,                                    /* Channel ID             */
     dim,                                    /* Dimension of cube      */
     i,                                      /* Iteration Counter      */
     ignore,                                 /* Place holder....       */
     rep_count,                              /* Iteration Counter      */
     reps,                                   /* Repetitions            */
     type,                                   /* Message Type           */
     cnt,                                    /* Msg Byte Count         */
     node,                                   /* Sending Node           */
     pid;                                    /* Process ID             */
```

```
long max_first,                                 /* Max time- 1st answer  */
     min_first,                                 /* Min time- 1st answer  */
     total_first,                               /* Total time (for avg)  */
     max_all,                                   /* Max time- all answers */
     min_all,                                   /* Min time- all answers */
     total_all,                                 /* Total time (for avg)  */
     first[32],                                 /* Time to first answer  */
     temp_first,                                /* Temp first answer     */
     all;                                       /* Time to all answers   */


/*----- Get the Problem Parameters from the Terminal -----*/

     printf("Enter Board Size: ");
     scanf("%d",&board_size);
     printf("Enter Cube Dimension: ");
     scanf("%d",&dim);
     number_of_nodes = 1<<dim;
     printf("Enter Number of Repetitions: ");
     scanf("%d",&reps);


/*----- Initialize Time Data Variables -----*/

     max_first   = 0;
     min_first   = INFINITY;
     total_first = 0;
     max_all     = 0;
     min_all     = INFINITY;
     total_all   = 0;


/*----- Run the N-queens for 'reps' number of times....
     - load the iPSC with the Worker and the Control Process
     - Down load the Board Size and Number of Nodes in the cube
     - Wait for a FINISHED message from the Control Process
     - Collect the Timing Data    -----*/

     for(rep_count=1; rep_count<=reps; rep_count++){
         cid = copen(HOST_PID);
         load("worker",-1,NODE_PID);
         lkill(0,NODE_PID);
         load("control",0,NODE_PID);

         for(i=0; i<number_of_nodes; i++)
             sendmsg(cid,BOARD_SIZE,&board_size,sizeof(board_size),i,
                     NODE_PID);

         sendmsg(cid,NUM_NODES,&number_of_nodes,sizeof(number_of_nodes),
                 0,NODE_PID);
```

```
            recvmsg(cid,&type,&ignore,sizeof(ignore),&cnt,&i,&pid);

            for(i=1; i<number_of_nodes; i++){
                sendmsg(cid,TIMES,&ignore,sizeof(ignore),i,NODE_PID);
                recvmsg(cid,&type,&first[i],sizeof(first[i]),&cnt,
                        &ignore,&pid);
            }

            sendmsg(cid,TIMES,&ignore,sizeof(ignore),0,NODE_PID);
            recvmsg(cid,&type,&all,sizeof(all),&cnt,&ignore,&pid);

            if (min_all > all) min_all = all;
            if (max_all < all) max_all = all;

            total_all  += all;
            temp_first = min(first);

            if (min_first > temp_first) min_first = temp_first;
            if (max_first < temp_first) max_first = temp_first;

            total_first += temp_first;
            lwaitall(-1,NODE_PID);
        }

    /*----- Compute average time to first solution and average time to
        all solutions                  -----*/

        total_first = total_first - min_first - max_first;
        total_all   = total_all - min_all - max_all;

        printf("Time to 1st Solution:%f\n",
                (float)total_first/((reps-2)*1000.0));
        printf("Time to All Solutions: %f\n",
               (float)total_all/((reps-2)*1000.0));

        cclose(cid);
    }
```

```
/************************************************************
*                                                           *
*   Name         : Min                                      *
*   Parameters   : f = Array of Long Integers               *
*   Function     : Find the minimum value in array 'f'      *
*                                                           *
************************************************************/
long min(f)
long f[32];
{
int i;
long m;

     m = INFINITY;
     for(i=1; i<number_of_nodes; i++)
          if ((m>f[i]) && (f[i] != 0))
               m = f[i];
     return(m);
}
```

The second program is the iPSC Control Process. The source code is on the following seven pages. Once loaded by the intermediate host program, the Control Process begins its initialization. First, the clock is started, a communications channel is opened, and the problem queue is initialized. Then receive the board size of the problem and the number of nodes in the cube. Next, create the initial set of problems to solve. This translates to generating the first levels of the search tree. Once the initial set of jobs is created, measure the start up time. Next, enter the control loop. As long as problems are in the queue and work has been assigned to the worker nodes, do the following, (1) check for a work request, and (2) try to hand out problems. If a work request comes in, set the status of the node to AVAILable. To hand out a problem two conditions must be met, (1) the queue must not be not empty and (2) a free processor is AVAILable. The control process terminates when the queue is empty and all workers have posted work requests. This translates to a machine state where no problems are available and all workers need a problem to solve. Upon exiting from the control loop, set a stop time and send a KILL message to all of the worker processes. Finally, send the timing data to the intermediate host.

```
/*********************************************************************
*                              THESIS                                *
*                                                                    *
*                  iPSC N-queens Control Process                     *
*                                                                    *
*                                                                    *
*  Date : 24 Oct 1986                                                *
*                                                                    *
*  Functions:  This C Language program defines the Control           *
*       Process for the iPSC Parallel N-queens Problem.              *
*       First, the initial problem information, Board                *
*       Size and Number of Nodes in the cube, are                    *
*       received from the Intermediate Host.  Then,                  *
*       the Controller creates an initial set of                     *
*       problems to solve.  This corresponds to                      *
*       generating the first levels of the search tree.              *
*                                                                    *
*       Now that the problem list is ready, get 'work                *
*       requests' from the Worker Processes and hand                 *
*       out problems.  Once the problem queue is empty               *
*       and all Worker Processes have posted a 'work                 *
*       request' messages, then all solutions have been              *
*       found.                                                       *
*                                                                    *
*  Language: C Language                                              *
*                                                                    *
*  Libraries:  iPSC Node definitions                                 *
*              FIFO Queue Routines                                   *
*                                                                    *
*  iPSC Operating System: Release 2.1                                *
*                                                                    *
*  Messages from Host:  Board Size                                   *
*                       Number of Nodes                              *
*                                                                    *
*  Message to Host:     Time to find all Solutions                   *
*                                                                    *
*  Message from Worker: Work Request                                 *
*                                                                    *
*  Messages to Worker:  Problem (subtree) to Solve                   *
*                       Termination (Kill) Message                   *
*                                                                    *
*  Author: Capt Rick Mraz                                            *
*                                                                    *
*                                                                    *
*********************************************************************/
```

```
/******************************************
*           HEADER FILES                  *
******************************************/
#include "/usr/ipsc/lib/cnode.def"        /* Hypercube Node Header   */
#include "q.h"                            /* Queue Routine Header    */

/******************************************
*        CONSTANT DEFINITIONS             *
******************************************/
#define AVAIL             -1              /* Node Available for work */
#define BUSY              -2              /* Node is busy             */
#define MAX_DIMENSION     5               /* Max dim of cube          */
#define TRUE              1               /* True  = 1                */
#define FALSE             0               /* False = 0                */

#define HOST              0x8000          /* Intermediate Host        */
#define NODE_PID          0               /* Node Process ID          */
#define HOST_PID          1               /* Host Process ID          */

#define BOARD_SIZE        10              /* Board Size Msg           */
#define ALL               30              /* Live Node Msg            */
#define WORK_REQUEST      45              /* Worker needs work...    */
#define E_NODE            50              /* Generate Children Msg   */
#define KILL              60              /* Kill worker Process      */
#define TIMES             100             /* Send Time information   */
#define FINISHED          120             /* Send Finished msg        */
#define NUM_NODES         130             /* Receive Cube Dimension  */

/******************************************
*             Global Variables            *
******************************************/

int  node_status[(1<<MAX_DIMENSION)+1],   /* Worker Status Array     */
     vector[VECTOR_SIZE+1],               /* Solution Vector          */
     number_of_nodes;                     /* Number of Nodes in cube */
```

```
/***********************************************
*      iPSC Control Process - Main Program     *
***********************************************/
main()
{
/*----- Control Process Variables -----*/

int  board_size,                        /* Size of Playing board      */
     i,                                 /* Iteration Counter          */
     ignore,                            /* Place holder....           */
     next_node,                         /* Node that needs work       */
     request,                           /* Work Request Msg Var       */
     row,                               /* Place queen on this row    */
     work_assigned,                     /* Num of Workers Busy        */

     E_node[VECTOR_SIZE + 1],           /* Next node to expand        */
     cid,                               /* Channel ID                 */
     cnt,                               /* Msg Length                 */
     node,                              /* Node number                */
     pid;                               /* Process ID                 */

long start,                             /* Start timer                */
     stop,                              /* Stop timer                 */
     startup_time;                      /* Measure Start Up Time      */


/*----- Initialize the Control Process
     - Take a Time Stamp
     - Open a Communications Channel for the NODE Process ID
     - Initialize the queue         -----*/

     start = clock();
     cid   = copen(NODE_PID);
     q_init();


/*---- Receive the Board Size and the Number of Nodes from the Host----*/

     recvw(cid,BOARD_SIZE,&board_size,sizeof(board_size),&cnt,&node,
          &pid);
     recvw(cid,NUM_NODES,&number_of_nodes,sizeof(number_of_nodes),&cnt,
          &node,&pid);

/*----- Set all Worker Nodes to BUSY  -----*/

     work_assigned = number_of_nodes - 1;
     for(i=1; i<number_of_nodes; i++)
         node_status[i] = BUSY;
```

```
/*----- Generate the initial set of problems to solve....
        - generate the search tree until the number of problems (length
          of the queue) equal 5 times the number of nodes in the cube.
          This is optimized for large problems on large cube sizes.

        - Make sure to only include those jobs that meet the bound
          condition   -----*/

     vector[0] = 1;
     while ((q_length <= (number_of_nodes*5)) &&
            (vector[0] < board_size)){
        deleteq(vector);
        row = vector[0];
        vector[0]++;

        while (vector[row] <= board_size){
           vector[row]++;

           while ((vector[row] <=board_size) && (bound(row) == FALSE))
                  vector[row]++;

           if (vector[row] <= board_size)
             addq(vector);
        }
     }

     startup_time = clock() - start;

/*----- Enter the Control Loop
     - When the queue is empty (no more problems to solve)
     - and when no work has been assigned
     - then all solutions have been found      -----*/

     while ((q_status != Q_EMPTY) || (work_assigned)){

     /*---- If the queue is not empty and if there is a free
            processor, then get an E-Node (problem) from the queue
            and send it to the Worker Node  -----*/

        while ((q_status != Q_EMPTY) &&
               ((next_node = get_free_processor()) != BUSY)){
                deleteq(E_node);
                node_status[next_node] = BUSY;
                work_assigned++;
                sendw(cid,E_NODE,E_node,sizeof(E_node),
                      next_node,NODE_PID);
        }
```

```
        /*----- If a 'work request' comes in, update the node status
                 to AVAILable   -----*/

           while (probe(cid,WORK_REQUEST) >=0){
              recvw(cid,WORK_REQUEST,&request,sizeof(request),&cnt,&node,
                   &pid);
              work_assigned--;
              node_status[node] = AVAIL;
           }
        }

    /*----- Once all solutions have been found,
        - take time stamp
        - notify the Intermediate Host that the computation is complete
        - send a KILL message to all Worker Processes
        - and then send the computation time data to the Host -----*/

        stop = clock() - start;

        sendw(cid,TIMES,&i,sizeof(i),HOST,HOST_PID);
        for(i=1; i<number_of_nodes; i++)
            sendw(cid,KILL,&ignore,sizeof(ignore),i,NODE_PID);

        recvw(cid,TIMES,&ignore,sizeof(ignore),&cnt,&node,&pid);
        sendw(cid,ALL,&stop,sizeof(stop),HOST,HOST_PID);

        cclose(cid);
    }


    /*********************************************************
    *                                                        *
    *  Name         : Get Free Processor                     *
    *  Parameters   : None                                   *
    *  Function     : Return the number of the free          *
    *                 processor, or return BUSY is all       *
    *                 processors are busy.                   *
    *                                                        *
    *********************************************************/
    get_free_processor()
    {
        int i;

        for(i=1; i<number_of_nodes; i++)
           if (node_status[i] == AVAIL)
              return(i);

        return(BUSY);
    }
```

```
/*********************************************************
*                                                        *
*  Name         : Bound                                  *
*  Parameters   : Row = the row to place the queen on    *
*  Function     : Make sure the queen is not on the      *
*                 same column or on a similar diagonal   *
*                 with another queen.                    *
*                                                        *
*********************************************************/
bound(row)
int row;
{
     int i;

     for(i=1; i<row; i++)
        if ((vector[i] == vector[row]) ||             /* same column   */
         (abs(vector[i]-vector[row]) == abs(i-row)))   /* same diagonal */
               return(FALSE);

     return(TRUE);
}
```

```c
/******************************************************************
*                          THESIS                                 *
*                                                                 *
*            iPSC N-queens Queue Routine Header                   *
*                                                                 *
*                                                                 *
*  Date : 24 Oct 1986                                             *
*                                                                 *
*  Function: This header file defines the FIFO queue routines    *
*            that the iPSC Control Process needs to maintain      *
*            a list of live nodes (list of problems to solve).    *
*            For this implementation, a simple 'ring' data-       *
*            structure was used for the FIFO queue.               *
*                                                                 *
*  Language: C Language                                           *
*                                                                 *
*  iPSC Operating System:  Release 2.1                            *
*                                                                 *
*  Author: Capt Rick Mraz                                         *
*                                                                 *
******************************************************************/

#define Q_BUSY          1       /* The Queue is being used    */
#define Q_EMPTY         2       /* The Queue is Empty         */
#define Q_FULL          3       /* The Queue is Full          */
#define Q_SIZE          200     /* Max of 200 elements        */
#define VECTOR_SIZE     25      /* Max Problem = 25-queens    */

int  front,                          /* Front pointer to the queue*/
     q[Q_SIZE][VECTOR_SIZE+1],       /* Queue definition          */
     q_length,                       /* Queue Length              */
     q_status,                       /* Queue Status={Q_EMPTY,
                                          Q_BUSY, Q_FULL}         */
     rear_q;                         /* End of the queue pointer  */

int  addq(),                         /* Add queue function        */
     deleteq();                      /* Delete queue function     */
```

The next source code listing is the worker process. It resides on all nodes except node 0. The worker process is the sequential version of the N-queens problem modified with some communications to get the initial problem size and to interact with the control process. Once loaded by the intermediate host program, the worker process initializes the answer_count and opens a communications channel. Then it receives the board size from the host and sends a work request to the controller. Once the work request is sent, the worker process enters an infinite control loop. Inside this loop, the worker waits for two events, (1) receive an E_Node message, and (2) receive a KILL message. If an E_Node message arrives, then find all solutions in the subtree defined by the E-node. This portion of the code is the sequential N-queens. The boundary condition on the "while" loop was changed to only examine the subtree instead of the entire search space. Remember to take a time hack after finding the first answer. Upon finding all solutions in the subtree, send a work request to the controller. If a KILL message arrives, then terminate the infinite loop, and send the timing data to the host.

```
/*****************************************************************
*                          THESIS                               *
*                                                               *
*              iPSC N-queens Worker Process                     *
*                                                               *
*                                                               *
*  Date : 24 Oct 1986                                           *
*                                                               *
*  Functions:  This C Language program defines the Worker       *
*        Process for the iPSC Parallel N-queens Problem.        *
*        First, the initial problem information, Board          *
*        Size, is received from the Intermediate Host.          *
*        Then, a 'work request' is sent to the Control          *
*        Process (Node 0).                                      *
*                                                               *
*        Upon entering an Infinite Loop, wait for an            *
*        E_Node (Problem) Message or a Terminate (Kill)         *
*        message from the Control Process.  If an E_Node        *
*        message arrives, then find all answers in that         *
*        subtree.  If a Kill message arrives, then              *
*        terminate the Infinite Loop.                           *
*                                                               *
*        Upon terminating the Infinite Loop, send run           *
*        time data back to the Host.                            *
*                                                               *
*  Language: C Language                                         *
*                                                               *
*  Libraries:  iPSC Node definitions                            *
*                                                               *
*  iPSC Operating System:  Release 2.1                          *
*                                                               *
*  Message from Host:  Board Size                               *
*                                                               *
*  Message to Host:    Time to find first Solution              *
*                                                               *
*  Message from Control: Solve Subtree (E_node)                 *
*                        Termination (Kill) Message             *
*                                                               *
*  Message to Control:   Work Request                           *
*                                                               *
*  Author: Capt Rick Mraz                                       *
*                                                               *
*                                                               *
*****************************************************************/
```

```
/******************************************
*           HEADER FILES                  *
******************************************/
#include "/usr/ipsc/lib/cnode.def"       /* iPSC Node Definitions     */


/******************************************
*        CONSTANT DEFINITIONS             *
******************************************/
#define TRUE               1              /* True  = 1                 */
#define FALSE              0              /* False = 0                 */

#define CONTROLLER         0              /* Controller = Node 0       */
#define HOST               0x8000         /* Intermediate Host         */
#define NODE_PID           0              /* Node Process ID           */
#define HOST_PID           1              /* Host Process ID           */

#define BOARD_SIZE         10             /* Board Size Msg            */
#define FIRST              20             /* Send time to 1st answer   */
#define WORK_REQUEST       45             /* Worker needs work...      */
#define E_NODE             50             /* Generate Children Msg     */
#define KILL               60             /* Kill worker Process       */
#define TIMES              100            /* Send Time information     */


/******************************************
*             Global Variables            *
******************************************/
int  vector[VECTOR_SIZE+1];              /* Solution Vector           */


/**********************************************
*        Worker Process - Main Program        *
**********************************************/
main()
{
/*----- Worker Process Variables -----*/

int  answer_count,                       /* Count the Answer Nodes    */
     board_size;                         /* Size of playing board     */
     first,                              /* Solving First Solution    */
     i,                                  /* Iteration Counter         */
     ignore,                             /* Place holder....          */
     me,                                 /* My node number            */
     root,                               /* Root of the Subtree       */
     row,                                /* The Queen goes here       */
     cid,                                /* Channel Id                */
     cnt,                                /* Msg Length Count          */
     node,                               /* Node msg goes to          */
     pid;                                /* Process ID                */
```

```
long    start,                          /* Start Time Hack          */
        first_answer;                   /* Time to first Answer     */


/*----- Initialize the Worker Process
     - Set the first answer flag to TRUE
     - Find out my node number
     - Set the answer count to 0
     - Open a Communications Channel for the NODE Process ID -----*/


     first = TRUE;
     me    = mynode();
     count = 0;
     cid   = copen(NODE_PID);

/*----- Receive the Board Size and Send a Work Request  -----*/

     recvw(cid,BOARD_SIZE,&board_size,sizeof(board_size),
           &cnt,&node,&pid);
     start = clock();
     sendw(cid,WORK_REQUEST,&me,sizeof(me),CONTROLLER,NODE_PID);

/*----- Enter an Infinite Loop Waiting for problems to solve -----*/

     for(;;){

     /*----- If an E_Node arrives,
         - Receive the messge
         - Solve the Subtree for all Answer Nodes
         - Once finished, send a 'work request' the the Controller -----*/

         if (probe(cid,E_NODE) >= 0){
               recvw(cid,E_NODE,vector,sizeof(vector),&cnt,&node,&pid);

         /*----- Place the queen in this row and subtree root -----*/
               row  = vector[0];
               root = row-1;

         /*----- Don't backtrack past the root of the subtree -----*/
               while (row>root){

         /*----- Place the next queen on the board -----*/
                  vector[row] = vector[row] + 1;
```

```
        /*----- Check for the bound conditions -----*/
                while ((vector[row] <= board_size) &&
                        (bound(vector,row) == FALSE))
                         vector[row] = vector[row] + 1;

          /*----- If the last queen is successfully put on the board,
                - then an answer node has been found.  If it is the
                  first answer, then take a time check.
                - If the queen is less than the board size, then
                  increment the row variable so the next queen can be
                  placed on the board
                - If the queen is greater than the board size, then
                  a node has been reached with no answers in the
                  subtree.

                  Continue the search by 'backtracking', (i.e.
                  decrement the row variable.  This will cause the
                  problem solver to find the next valid position for
                  the previous queen. -----*/

                if (vector[row] <= board_size){
                   if (row == board_size){
                       if (first){
                           first_answer = clock() - start;
                           count++;
                           first = FALSE;
                       }
                  else
                      count++;
                   }
                   else{
                  row++;
                       vector[row] = 0;
                   }
                else
                   row--;
                }

        /*----- After all Answers are found in the Subtree, send a
               'work request' to the Control Process -----*/

               sendw(cid,WORK_REQUEST,&me,sizeof(me),CONTROLLER,NODE_PID);
               }


        /*----- During the search, if a Kill message is received, break
                from the Infinite Loop -----*/

               if (probe(cid,KILL) >= 0) break;
           }
```

```
/*----- Once the terminate message arrives,
        - then send the computation time data to the Host -----*/

        recvw(cid,TIMES,&ignore,sizeof(ignore),&cnt,&node,&pid);
        sendw(cid,FIRST,&first_answer,sizeof(first_answer),
              HOST,HOST_PID);
}


/*********************************************************
*                                                        *
*  Name        : Bound                                   *
*  Parameters  : Row = Queen is placed in this row       *
*  Function    : Make sure the new queen is not in the   *
*                same column as any other queens, and    *
*                cannot attack on a diagonal.            *
*                                                        *
*********************************************************/
bound(vector,row)
int vector[VECTOR_SIZE+1];
int row;
{
   int  i;

        for(i=1; i<row; i++)
            if ((vector[i] == vector[row]) ||
               (abs(vector[i]-vector[row]) == abs(i - row)))
                return(FALSE);

        return(TRUE);
}
```

In this final section, a trace of the parallel N-queens shows how the control and worker processes communicate to solve the N-queens problem. The example shown here is the 4-queens problem solved on a 2-dimension cube (4 nodes). The control process resides on Node 0 and the worker processes reside on Nodes 1, 2, and 3. The description of the parallel search starts after the intermediate host loaded the iPSC and each node performed its own initialization.

Live Node Q	Worker #1	Worker #2	Worker #3
(1,•,•,•) (2,•,•,•) (3,•,•,•) (4,•,•,•)	available	available	available
-- Solve Subtree (1,•,•,•) with Worker #1			
(2,•,•,•) (3,•,•,•) (4,•,•,•)	busy	available	available
-- Solve Subtree (2,•,•,•) with Worker #2			
(3,•,•,•) (4,•,•,•)	busy	busy	available
-- Receive a Work Request from Worker #1 -- Solve Subtree (3,•,•,•) with Worker #3			
(4,•,•,•)	busy	busy	busy
-- Handle Work Request from Worker #1			
<Empty>	busy	busy	busy
-- Receive Work Request from Worker #2			
<Empty>	busy	available	busy

-- Receive Work Request from Worker #3
-- Receive Work Request from Worker #1

<Empty>	available	available	available

-- Control Process send KILL messages to all Workers

-- Workers and Control Processes send timing information back to Host

<<< END TRACE >>>

Appendix B:

Sequential N-queens Source Code Description

The listing on the following pages was run on a DEC VAX 11/785 and an Elxsi System 6400. The run times for various board sizes sets a baseline for performance comparisons with the iPSC Hypercube (see Chapter VI, Performance Measures and Experiment Results). This version of the N-queens is written in C Language. The description of the code follows.

Page one of the listing defines the Header Files, Constants, and Variables used in the program. The main program starts on the second page of the listing. First, the initial state of the problem is set. The first solution flag is set to TRUE. This flag is used to take a time check when the first answer arrives. Next, the solution vector, $(x_1,x_2,\ldots,x_n)$ where n = the board size of the problem, is set to (0,•,...,•) where • defines an x_i that has not been determined. Then, the row variable is set to place the first queen on the board. After the user enters a board size, the 'start-time' is taken.

Next, all solutions are computed inside the while loop. Once the problem backtracks past row 1, the problem is finished. First, place the queen on the next column. The solution vector now looks like this, (1,•,...,•). Next, make sure this is a valid position by checking the bound function. The bound function makes sure that the new queen is not on the same column as any other queen and it checks that the new queen cannot attack on the diagonal. If the position is OK, then continue. If the position is not OK, then place the queen on the next column, (2,•,•,•), and check the bound again. If the queen is placed off the board (row > board_size), then continue.

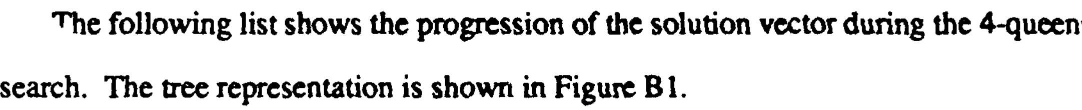
Once a valid solution vector is computed, the vector constitutes an answer if the row equals the board size. For this problem, the only solution vector of interest is the first one. Once the first solution arrives, take a time check. If the row does not equal the board size, then place the next queen on the board (increment the row variable). If the the row is greater than the board size, then an invalid state has been found and the solution vector must backtrack finding a better place for the previous queen (decrement row).

Once all of the solutions have been found, take a stop time and print the computation time to find the first solution and to find all solutions.

```
/*****************************************************************************
*                                  THESIS                                    *
*                                                                            *
*                       Sequential N-queens Problem                          *
*                                                                            *
*                                                                            *
*  Date:       8 Aug 1986                                                    *
*                                                                            *
*  Functions:  The following program solves the N-queens problem.            *
*              Given a 'board size', the program posts the time to           *
*              compute the first solution and the time to compute all        *
*              solutions.                                                    *
*                                                                            *
*  Language:   C Language                                                    *
*                                                                            *
*  Operating System: 4.2 BSD UNIX                                            *
*                                                                            *
*  Libraries:  Standard I/O      Types.h       Times.h                       *
*                                                                            *
*  Author :    Capt Rick Mraz                                                *
*                                                                            *
*                                                                            *
*****************************************************************************/
/***********************************************
*          HEADER FILES                        *
***********************************************/
#include <stdio.h>
#include </sys/h/types.h>
#include </sys/h/times.h>

/***********************************************
*          CONSTANTS                           *
***********************************************/
#define TRUE      1
#define FALSE     0

/***********************************************
*          N-QUEENS VARIABLES                  *
***********************************************/

int         first,                       /* Solving First Solution       */
            row,                         /* Place the Queen in the row   */
            board_size;                  /* Size of playing board        */

int         vector[12];                  /* Solution Vector              */

struct tms time_first;                   /* Time for First Solution      */
struct tms start_time;                   /* Start of Computation         */
struct tms stop_time;                    /* End of Computation           */
```

```
/***********************************************
* Main program for the N-queens problem        *
***********************************************/
main()
{
/*-----  Initialize the system -----*/

            first = TRUE;                   /* Find first solution          */
            vector[1] = 0;                  /* Initial Problem Vector       */
            row  = 1;                       /* Place 1st queen here....     */

            printf("Enter Board Size: ");   /* Get Board Size               */
            scanf("%d",&board_size);

            times(&start_time);             /* Set Start Time               */


/*-----     Find all Solutions -----*/

            while (row > 0){
             vector[row] = vector[row]+1;   /* Next Column                  */

/*----- Find a valid Column for the next queen -----*/

                while ((vector[row] <= board_size) &&
                         (bound(row) == FALSE))
                          vector[row] = vector[row] + 1;


/*---- If the first solution is found, then take a time stamp
*           otherwise,
*           place the next queen on the board
*       if a valid place for the queen can not be found, then 'backtrack' */

                if (vector[row] <= board_size){
                    if ((row == board_size) &&
                         (first)){
                          times(&time_first); /* Time for 1st solution      */
                          first = FALSE;
                    }
                else{
                    row++;                  /* Place next queen             */
                    vector[row] = 0;
                }
                else
                    row--;
                                            /* Backtrack                    */
            }/* end while loop */
```

```
/*----- Computation Over...Calculate the run times -----*/
            times(&stop_time);

            printf("Time First Solution: %f sec\n",

(float)(time_first.tms_utime-start_time.tms_utime)/60.0);
            printf("Time All Solutions: %f sec\n",

(float)(stop_time.tms_utime-start_time.tms_utime)/60.0);
}


/*******************************************************************
*                                                                  *
*  Name:         Bound                                             *
*                                                                  *
*  Parameters:   Row = Queen is placed in this row                 *
*                                                                  *
*  Function:     Make sure the new queen is not in the same        *
*                column as any other queens, and cannot attack     *
*                on a diagonal.  Return FALSE if the position is   *
*                not valid.  Return TRUE is the board position is  *
*                valid.                                            *
*                                                                  *
*******************************************************************/

bound(row)
int row;
{
   int  i;

     for(i=1; i<row; i++)
         if ((vector[i] == vector[row]) ||
              (abs(vector[i]-vector[row]) == abs(i - row)))
                   return(FALSE);

     return(TRUE);
}
```



The following list shows the progression of the solution vector during the 4-queens search. The tree representation is shown in Figure B1.

Vector	Notes
(0,•,•,•)	Initial Vector
(1,•,•,•)	Place the first queen in column 1
(1,1,•,•)	Place next queen...Same column (Bound)
(1,2,•,•)	Attack on Diagonal (Bound)
(1,3,•,•)	The first valid position for queen #2
(1,3,1,•)	Place next queen...Same column (Bound)
(1,3,2,•)	Attack on Diagonal (Bound)
(1,3,3,•)	Same column (Bound)
(1,3,4,•)	Attack on Diagonal (Bound)
(1,3,5,•)	The third queen doesn't have a valid position
(1,4,•,•)	Backtrack- find the next valid position for queen #2
(1,4,1,•)	Same column (Bound)
(1,4,2,•)	The first valid position for queen #3
(1,4,2,1)	Same column (Bound)
(1,4,2,2)	Same column (Bound)
(1,4,2,3)	Attack on Diagonal (Bound)
(1,4,2,4)	Same column (Bound)
(1,4,2,5)	The fourth queen doesn't have a valid position
(1,4,3,•)	Backtrack- find the next place for queen #3...Attack on Diagonal (Bound)
(1,4,4,•)	Same column (Bound)

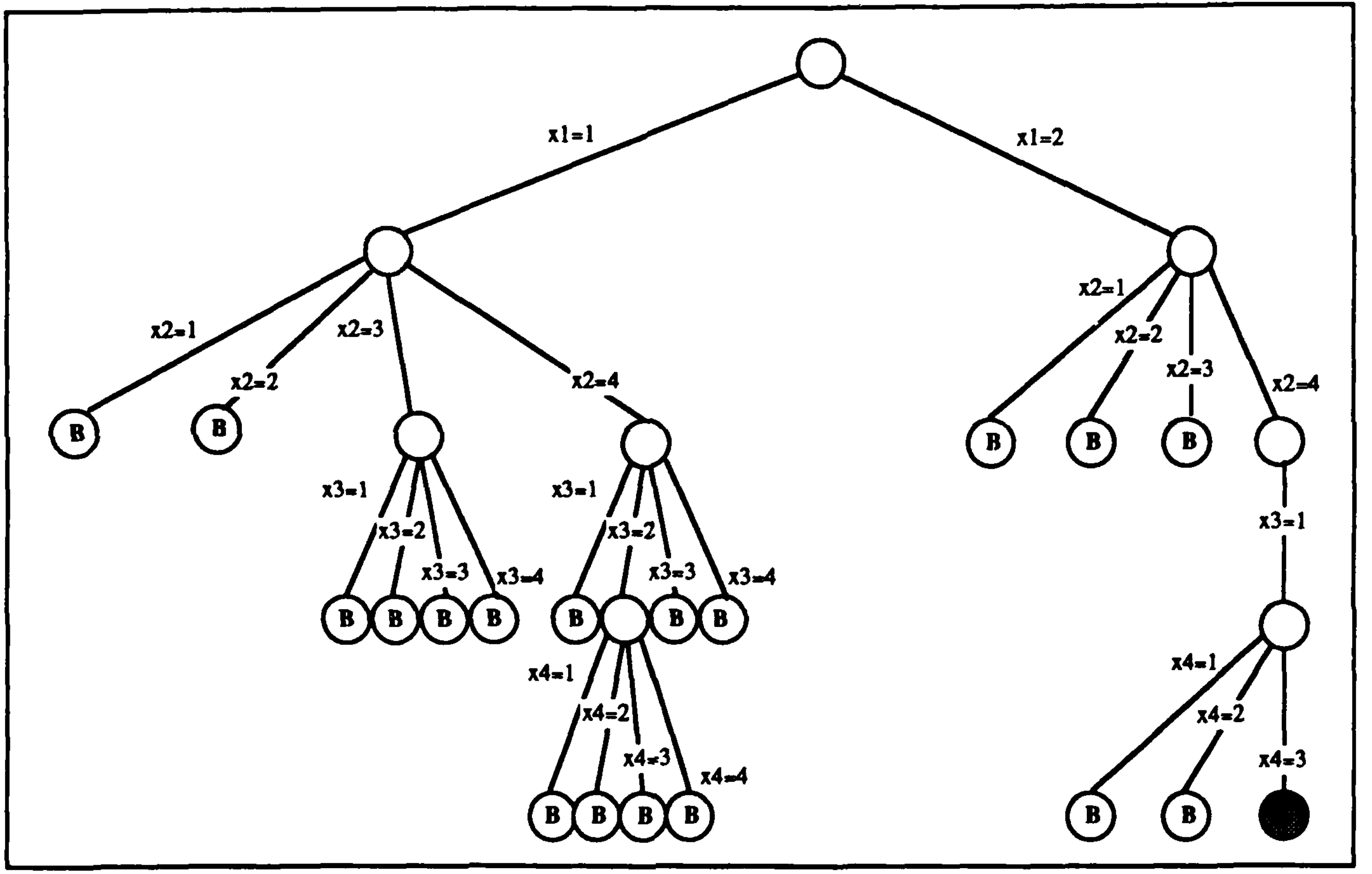


Figure 36: Portion of 4-queens Solution Space
Generated During Search
(16:331)

Vector	Notes
(1,4,5,•)	The third queen doesn't have a valid position
(1,5,•,•)	Backtrack- The second queen doesn't have a valid position
(2,•,•,•)	Backtrack- The next valid position for queen #1
(2,1,•,•)	Attack on Diagonal (Bound)
(2,2,•,•)	Same column (Bound)
(2,3,•,•)	Attack on Diagonal (Bound)
(2,4,•,•)	The first valid position for queen #2
(2,4,1,•)	The first valid position for queen #3
(2,4,1,1)	Same column (Bound)
(2,4,1,2)	Same column (Bound)
(2,4,1,3)	**The first answer to the 4-queens problem**

Appendix C:

iPSC Deadline Job Scheduling Source Code Description

The parallel source code for the deadline job scheduling problem is on the following pages. This description is divided into three sections, (1) intermediate host program, (2) control process program, and (3) worker process program. This software is written in C Language for the Intel iPSC Hypercube computer.

First, a description of the intermediate host source code. This program serves as an interface between the user and the iPSC. Therefore, information about the dimension of hypercube, number of jobs to schedule, as well as the 3-tuples that define each job must be provided to this program. Upon reading this data, the host program initializes the hypercube by down loading the control and the worker software into the appropriate nodes. Then, the number of jobs to schedule and the job 3-tuple data is loaded into the cube. After this initialization, the intermediate host waits for an answer from the control process. Once this message is received, then the best answer and the run time information is retrieved. The host listing is on the next few pages.

```
/*******************************************************************
*                          THESIS                                  *
*              iPSC Deadline Job Scheduling                        *
*                Intermediate Host Program                         *
*                                                                  *
*                                                                  *
*  Date:  5 Nov 1986                                               *
*                                                                  *
*  Functions:  This is the iPSC intermediate host program for     *
*         the the Deadline Job Scheduling Problem.  This program *
*         loads the iPSC with the Worker and Control Processes,  *
*         and then waits for an answer message from the Control  *
*         Process.  Upon receiving this message, the Host        *
*         Program retrieves the Timing Data.                     *
*                                                                  *
*         The averaging method to compute the time information   *
*         is quite simple.  First, run the problem for 'n'       *
*         repetitions (a user entered parameter).  Next, remove  *
*         the maximum time as well as the minimum time and       *
*         average the remaining numbers.  It should be noted     *
*         that the minimum number of repetitions that this       *
*         program should run is 3.                               *
*                                                                  *
*         The following information must be available from the   *
*         standard input device so this program can initialize   *
*         the iPSC processing elements:                          *
*                                                                  *
*         1- iPSC Dimension                                       *
*         2- Number of Repetitions to run                         *
*         3- Number of Jobs to Schedule                           *
*         4- (Penalty, Deadline, Time)  triple defined            *
*            for each of the jobs                                 *
*                                                                  *
*  Libraries: Standard Input & Output                             *
*             iPSC chost definitions                              *
*                                                                  *
*  Language: C Language                                           *
*                                                                  *
*  Operating System: iPSC Release 2.1                             *
*                                                                  *
*  Input Parameters: Dimension of the iPSC                        *
*                    Number of Repetitions                         *
*                    Number of Jobs to Schedule                    *
*                    (Penalty, Deadline, Time) Job ID 3-tuple      *
*                                                                  *
*  Outputs: Start Up Time                                          *
*           Total Run Time                                         *
*                                                                  *
*  Author : Capt Rick Mraz                                         *
*                                                                  *
*******************************************************************/
```

```c
/*****************************************************
*                  Header Files                     *
*****************************************************/
#include <stdio.h>                    /* Standard IO                  */
#include "job.h"                      /* Job Information Header       */
#include "q.h"                        /* Solution Vector Header       */
#include "/usr/ipsc/lib/chost.def"    /* iPSC Host Definitions        */


/*****************************************************
*                Define Constants                   *
*****************************************************/
#define INFINITY       9999           /* Need a big Number??          */
#define ALL_NODES      -1             /* All cube nodes               */
#define ALL_PIDS       -1             /* All Process IDs              */

#define DIM            30             /* Send Dimension msg           */
#define NUM_JOBS       50             /* Send number of jobs          */
#define NEW_JOB        60             /* Send a Job Triple            */
#define U_BOUND        70             /* Send an Upper Bound          */

#define NODE_PID       0              /* Node Process ID              */
#define CONTROLLER     0              /* Controller  = Node 0         */
#define HOST_PID       1              /* Host Process ID              */


/*****************************************************
*          Define Global Variables                  *
*****************************************************/
int dim,                              /* Cube dimension               */
    i,                                /* Iteration Counter            */
    ignore,                           /* dummy variable               */
    u_bound,                          /* Upper Bound                  */
    number_of_jobs,                   /* Number of Jobs               */
    number_of_nodes,                  /* Number of Nodes in Cube      */

    cid,                              /* Channel ID                   */
    node,                             /* Node number                  */
    cnt,                              /* Msg Length                   */
    type,                             /* Msg Type                     */
    pid;                              /* Process ID                   */

JOB job_list[MAX_JOBS];               /* Array of Job Triples         */

NODE best;                            /* Global Best Solution         */
```

```
/***********************************************************
*                      Main Program                        *
***********************************************************/
main()
{
int  cycle,                          /* Iteration Counter          */
     reps;                           /* Number of Repetitions      */

long run,                            /* Run Time                   */
     init,                           /* Start Up Time              */
     min_init,                       /* Min Init Time              */
     max_init,                       /* Max Init Time              */
     total_init,                     /* Total Init Time            */
     min_time,                       /* Min run time               */
     max_time,                       /* Max run time               */
     total_time;                     /* Total run time             */

     /*----- Get the Problem Parameters from the Terminal -----*/

     printf("\n\n\tParallel Deadline Job Schedule System\n");
     printf("\t-------------------------------------\n\n");

     scanf("%d",&dim);
     number_of_nodes = 1<<dim;

     scanf("%d",&reps);
     scanf("%d",&number_of_jobs);
     for(i=1; i<=number_of_jobs; i++)
     scanf("%d%d%d",&job_list[i].p,&job_list[i].d,&job_list[i].t);

     /*----- Initialize Time Data Variables -----*/

     cid     = copen(HOST_PID);
     u_bound = INFINITY;

     min_init   = 100000;
     min_time   = 100000;
     max_init   = 0;
     max_time   = 0;
     total_init = 0;
     total_time = 0;
```

```
/*----- Run the DJS Problem for 'reps' number of times....
        - load the iPSC with the Worker and the Control Process
        - Down load the Dimension of the Cube, the Number of Jobs,
          and the array of Job Triples.
        - Wait for a FINISHED message from the Control Process
        - Collect the Timing Data -----*/

for(cycle=1; cycle<=reps; cycle++){

        load("worker",ALL_NODES,NODE_PID);
        lkill(0,NODE_PID);
        load("control",0,NODE_PID);

        sendmsg(cid,DIM,&dim,sizeof(dim),0,NODE_PID);

        for(node=0; node<number_of_nodes; node++){
            sendmsg(cid,NUM_JOBS,&number_of_jobs,
                     sizeof(number_of_jobs),node,NODE_PID);
            sendmsg(cid,U_BOUND,&u_bound,
                     sizeof(u_bound),node,NODE_PID);
        }

        for(node=0; node<number_of_nodes; node++)
            for(i=1; i<=number_of_jobs; i++)
                sendmsg(cid,NEW_JOB,&job_list[i],sizeof(JOB),
                        node,NODE_PID);

        recvmsg(cid,&type,&best,sizeof(NODE),&cnt,&i,&pid);
        recvmsg(cid,&type,&init,sizeof(init),&cnt,&i,&pid);
        recvmsg(cid,&type,&run,sizeof(run),&cnt,&i,&pid);

        if (min_init > init) min_init = init;
        if (max_init < init) max_init = init;

        if (min_time > run) min_time = run;
        if (max_time < run) max_time = run;

        total_init += init;
        total_time += run;

        lwaitall(-1,NODE_PID);
}

/*--- Print the best job schedule once all reps have been run ---*/

printf("The Best Job Schedule is: {");
for(i=1; i<=number_of_jobs; i++)
        if (best.vector[i] == 1)
                        printf("%d ",i);
printf("}\n\n");
```

```
        i = penalty(best);
        printf("Penalty = %d\n",i);
        i = total_run(best);
        printf("Total Time = %d\n",i);
        i = max(best,number_of_jobs);
        printf("Deadline = %d\n\n",i);

        total_init = total_init - min_init - max_init;
        total_time = total_time - min_time - max_time;

        printf("Init Time   = %f (sec)\tTotal Time= %f (sec)\n",
        (float)total_init/((reps-2)*1000.0),
        (float)total_time/((reps-2)*1000.0));

        cclose(cid);
        lwaitall(-1,NODE_PID);
}


/*********************************************************
*                                                        *
*  Name       : Penalty                                  *
*  Parameters : n = Solution Node                        *
*  Function   : Return the total penalty of the node     *
*               defined by n.                            *
*                                                        *
*********************************************************/
penalty(n)
NODE n;
{
      int i,p;

      p = 0;
      for(i=1; i<=number_of_jobs; i++)
              if (n.vector[i] != 1)
                          p += job_list[i].p;

      return(p);
}
```

```
/*********************************************************
*                                                        *
*  Name       : Max                                      *
*  Parameters : E-node  = Solution Vector                *
*               Last    = Last Job in the Schedule       *
*  Function   : Return the Maximum Deadline of the       *
*               those Jobs Scheduled in the E-node.      *
*                                                        *
*********************************************************/
max(E_node,last)
NODE E_node;
int last;
{
     int i,m;

     m = job_list[last].d;
     for(i=1; i<=last; i++)
             if ((E_node.vector[i] == 1) &&
                 (m < job_list[i].d))
                          m = job_list[i].d;

     return(m);
}


/*********************************************************
*                                                        *
*  Name       : Total Run                                *
*  Parameters : n = solution vector                      *
*  Function   : Return the total run time of all jobs    *
*               scheduled in node n.                     *
*                                                        *
*********************************************************/
total_run(n)
NODE n;
{
     int i,t;

     t = 0;
     for(i=1; i<=number_of_jobs; i++)
             if (n.vector[i] == 1)
                          t += job_list[i].t;

     return(t);
}
```

```
/*****************************************************************
*                          Job Header File                       *
*                                                                *
*                                                                *
*                                                                *
*  Date: 28 Nov 86                                               *
*                                                                *
*  Functions: This header file describes the 3-tuple that        *
*             defines a Job for the job scheduling program.      *
*                                                                *
*                                                                *
*****************************************************************/
/************************************************
*  Node Information Record                      *
************************************************/
#define MAX_JOBS   50


/************************************************
*  Node Information Record                      *
************************************************/
struct job {
   int p;                                   /* Penalty           */
   int d;                                   /* Deadline          */
   int t};                                  /* Time to Run       */

typedef struct job JOB;                     /* Define the type job */
```

```
/*****************************************************************
*                        QUEUE HEADER FILE                       *
*                                                                *
*      Date: 28 Nov 1986                                         *
*                                                                *
*      Function: The following header file defines the solution  *
*                vector and the priority queue functions needed  *
*                by the parallel job scheduling system.          *
*                                                                *
*****************************************************************/
#include <stdio.h>

/***********************************************
*      Define Constants                        *
***********************************************/
#define TRUE          1            /* True is defined as integer 1   */
#define FALSE         0            /* False is defined as integer 0  */
#define VECTOR_SIZE   15           /* Vector Size                    */
#define Q_SIZE        600          /* Maximum Length of Queue        */
#define EOL           -1           /* End of Linked List Marker      */
#define EMPTY         1            /* Queue Empty (status)           */
#define Q_BUSY        2            /* Queue Busy  (status)           */
#define FULL          3            /* Queue Full  (status)           */


/***********************************************
*      Node Information Record                 *
***********************************************/
struct node {
       int vector[VECTOR_SIZE+1];  /* Solution Vector                */
       int cost;                   /* Cost of the solution vector    */
       int link};                  /* Forward Links for the queues   */

typedef struct node NODE;          /* Define the type NODE           */


int    front,                      /* Front of the queue pointer     */
       q_length,                   /* Queue Length                   */
       q_status,                   /* Queue Status                   */
       freeptr;                    /* Free List pointer              */

NODE   q[Q_SIZE+1];                /* Queue of search space nodes    */

NODE delete_q();                   /* Delete a node from the queue   */
insert_priority();                 /* Insert by priority             */
```

The second program needed to run the deadline job scheduling problem in parallel, is the control process. This is defined by the source code on the next pages. Once the intermediate host loads the control process into the hypercube, its initialization begins. First, the dimension of the hypercube, the number of jobs to schedule, the 3-tuples for each job, and the initial upper bound are received from the intermediate host. Then the control process creates the initial set of problems to solve. After this is finished, the control process enters a loop with three operations. The first operation is to monitor for 'work request' messages from the worker nodes. If a 'work request' arrives, then set the status of that node to AVAILable. The second operation is to hand out problems to solve. To hand out a problem the problem queue must not be empty and a worker must be AVAILable. Once a problem is sent to a worker, mark the node as BUSY. The third operation is to collect the 'local' best answers from the worker nodes. If the upper bound of a 'local' best answer is less than the present upper bound, then a new upper bound and a new global best answer has been found. Remember the new best answer two constraints and broadcast the new upper bound to all workers. This process terminates once the problem queue is empty and all workers have posted a 'work request'. This translates to a machine state where there are no problems to solve and all workers need work. Upon reaching this machine state, terminate the search by sending a KILL message to all workers and sending the best answer and run time data to the intermediate host.

```
/*********************************************************************
*                           THESIS                                   *
*                 iPSC Deadline Job Scheduling                       *
*                 iPSC Control Process Program                       *
*                                                                    *
*                                                                    *
*  Date:  5 Nov 1986                                                 *
*                                                                    *
*  Functions:  This is the iPSC Control Process program for the      *
*         Deadline Job Scheduling problem.  The control process      *
*         monitors the progress of the parallel search and           *
*         returns the answer to the intermediate host.  The          *
*         Control Process needs the following information to         *
*         conduct a parallel search:                                 *
*                                                                    *
*         1- iPSC Dimension                                          *
*         2- Number of Repetitions to run                            *
*         3- Number of Jobs to Schedule                              *
*         4- (Penalty, Deadline, Time)  triple defined               *
*            for each of the jobs                                    *
*                                                                    *
*         After receiving this information, the control process      *
*         creates the initial set of problems to solve.  Once        *
*         these problems are ready, then the controller enters a*
*         loop to solve the search.  Inside this loop, three         *
*         operations are performed, (1) Handle work requests         *
*         from the worker nodes, (2) Hand out problems to worker*
*         nodes, and (3) Maintain the global upper bound and         *
*         global best answer.                                        *
*                                                                    *
*         Once the problem queue is empty and all workers            *
*         have posted a work request, terminate the search by        *
*         sending a KILL message to all workers and by sending       *
*         the answer and the run time data to the intermediate       *
*         host.                                                      *
*                                                                    *
*  Libraries: Standard Input & Output                                *
*             iPSC cnode definitions                                 *
*                                                                    *
*  Language:  C Language                                             *
*                                                                    *
*  Operating System: iPSC Release 2.1                                *
*                                                                    *
*  Message from Host: Dimension of the iPSC                          *
*                     Number of Repetitions                          *
*                     Number of Jobs to Schedule                     *
*                    (Penalty, Deadline, Time) Job ID 3-Tuple        *
*                                                                    *
*  Message to Host: Answer to the Search                             *
*                   Start Up Time                                    *
*                   Total Run Time                                   *
```

```
*                                                                  *
*  Message to Worker: Solve Subtree (E_node)                       *
*                     Termination (Kill) Message                   *
*                                                                  *
*  Message from Worker: Work Request                               *
*                       Total Run Time                             *
*                                                                  *
*  Author : Capt Rick Mraz                                         *
*                                                                  *
*******************************************************************/

/***************************************************
*                 HEADER FILES                     *
***************************************************/
#include <stdio.h>                          /* Standard IO           */
#include "/usr/ipsc/lib/cnode.def"          /* Hypercube Node Header */
#include "q.h"                              /* Queue Routine Header  */
#include "job.h"                            /* Job information Header */

/***********************************************
*         Define Constants                     *
***********************************************/
#define AVAIL           -1          /* Node Available for work    */
#define BUSY            -2          /* Node is busy               */
#define MAX_DIMENSION   5           /* Max dim of cube            */

#define HOST            0x8000      /* Intermediate Host          */
#define NODE_PID        0           /* Node Process ID            */
#define CONTROLLER      0           /* Controller  = Node 0       */
#define HOST_PID        1           /* Host Process ID            */
#define INFINITY        9999        /* Need a big Number??        */

#define E_NODE          10          /* Generate Children Msg      */
#define DIM             30          /* Rec the cube dimension     */
#define WORK_REQUEST    40          /* Worker needs work...       */
#define NUM_JOBS        50          /* Rec number of jobs         */
#define NEW_JOB         60          /* Rec a Job triple           */
#define U_BOUND         70          /* Rec/Send the Upper Bound   */
#define KILL            80          /* Kill worker Process        */
#define TIMES           100         /* Send Time information      */
#define INIT            110         /* Send Start Up time         */
#define STOP            130         /* Send Stop time             */
#define ANSWER          160         /* Send the answer to Host    */
#define BEST            200         /* Rec a best from Workers    */
```

```
/**************************************************
*           Define Global Variables               *
**************************************************/
int   number_of_nodes,               /* Number of nodes in cube    */
      number_of_jobs,                /* Number of jobs             */
      u_bound,                       /* Global Upper Bound         */
      node_status[33];               /* Worker Status Array        */

NODE best,                           /* Global Best Solution       */
     local_best,                     /* Worker Best Solution       */
     copy_node(),                    /* Copy a node                */
     E_node;                         /* Next node to expand        */


JOB  job_list[MAX_JOBS];             /* Array of Job Triple Info   */


/**********************************************************
*                 Main Program                            *
**********************************************************/
main()
{
int   i,                             /* Iteration Counter          */
      ignore,                        /* Forget this parameter      */
      dim,                           /* Dimension of cube          */
      best_bound,                    /* Best Solution U Bound      */
      kill,                          /* Kill Msg Variable          */
      new_bound,                     /* U Bound- Local Best Node   */
      next_node,                     /* Node that needs work       */
      request,                       /* Work Request Msg Var       */
      work_assigned,

      cid,                           /* Channel ID                 */
      node,                          /* Node number                */
      cnt,                           /* Msg Length                 */
      pid;                           /* Process ID                 */

long  start,                         /* Start timer                */
      stop,                          /* Stop timer                 */
      init_time,                     /* Measure Start Up Time      */
      time_first;                    /* Time to first solution     */


/*----- Initialize the Control Process by....
      - Take a start time hack
      - Open a communications channel
      - Receive the dimension of the hypercube
      - Receive the Number of Jobs to Schedule
      - Receive the initial Upper Bound
      - Receive the Job Triple information for all jobs
      - Generate the Initial Set of Problems to Solve -----*/
```

```
        start = clock();
        cid   = copen(NODE_PID);

        recvw(cid,DIM,&dim,sizeof(dim),&cnt,&node,&pid);
        number_of_nodes = 1<<dim;

        work_assigned = number_of_nodes - 1;

        recvw(cid,NUM_JOBS,&number_of_jobs,sizeof(number_of_jobs),
        &cnt,&node,&pid);

        recvw(cid,U_BOUND,&u_bound,sizeof(u_bound),&cnt,&node,&pid);

        for(i=1; i<=number_of_jobs; i++)
             recvw(cid,NEW_JOB,&job_list[i],sizeof(JOB),&cnt,&node,&pid);

        q_init();
        create_init_jobs();

        best.cost = INFINITY;

        init_time = clock() - start;

   /*----- While there are problems to solve or work has been assigned,
           - Collect Work Requests from the Worker Nodes
           - Hand out problems to Solve
           - Collect Local Best answers and Maintain the Global Upper Bound
             and Global Best Answer  -----*/

           while((q_status != EMPTY) || (work_assigned)){

           /*--- Collect Work Requests and Set the Status to AVAILable ---*/

                while (probe(cid,WORK_REQUEST) >=0){
                    recvw(cid,WORK_REQUEST,&request,sizeof(request),
                           &cnt,&node,&pid);
                    node_status[node] = AVAIL;
                    work_assigned--;
                }
```

```
        /*----- If there are problems in the queue and if there is a
                 worker available,
              - Mark the Worker as BUSY
              - Get a problem (E-node) from the problem queue
              - Send it to the Worker -----*/

              while ((q_status != EMPTY) &&
                    ((next_node = get_free_processor()) != BUSY)){
                  node_status[next_node] = BUSY;
                  E_node = delete_q();
                  work_assigned++;
                  sendw(cid,E_NODE,&E_node,sizeof(NODE),next_node,
                        NODE_PID);
              }

        /*----- Collect the Local Best Answers from the Workers,
              - If the upper bound of the local best answer is less than
                the global upper bound, then a new global best answer
                has been found.
              - Remember the Global Best Solution and broadcast the new
                upper bound to all worker nodes -----*/

              while (probe(cid,BEST) >= 0){
                  recvw(cid,BEST,&local_best,sizeof(NODE),&cnt,
                        &node,&pid);

                  new_bound = bound(local_best);
                  if (new_bound < u_bound){
                      best = copy_node(local_best,best);
                      u_bound = new_bound;
                      for(i=1; i<number_of_nodes; i++)
                          sendw(cid,U_BOUND,&u_bound,sizeof(u_bound),
                                i,NODE_PID);
                  }
              }
        }

/*----- Once the best answer has been found, terminate the search by
        - taking a stop time
        - send a KILL message to all worker nodes
        - send timing information to the intermediate host -----*/

        stop = clock() - start;

        for(i=1; i<number_of_nodes; i++)
             sendw(cid,KILL,&ignore,sizeof(ignore),i,NODE_PID);
```

```
        sendw(cid,ANSWER,&best,sizeof(NODE),HOST,HOST_PID);
        sendw(cid,INIT,&init_time,sizeof(init_time),HOST,HOST_PID);
        sendw(cid,STOP,&stop,sizeof(stop),HOST,HOST_PID);

        cclose(cid);
}


/*********************************************************
*                                                        *
*  Name       : Get Free Processor                       *
*  Parameters : none                                     *
*  Function   : If a worker is available, return its     *
*               node number, otherwise return BUSY.      *
*                                                        *
*********************************************************/
get_free_processor()
{
     int i;

     for(i=1; i<=number_of_nodes; i++)
        if (node_status[i] == AVAIL)
             return(i);

     return(BUSY);
}

/*********************************************************
*                                                        *
*  Name       : Copy Node                                *
*  Parameters : n1 = Source Node                         *
*               n2 = Destination Node                    *
*                                                        *
*  Function   : Copy node, n1, into node, n2,  and       *
*               return node, n2.                         *
*                                                        *
*********************************************************/
NODE copy_node(n1,n2)
NODE n1,n2;
{
     int i;

     for (i=0; i<=VECTOR_SIZE; i++)
        n2.vector[i] = n1.vector[i];

     n2.cost = n1.cost;
     return(n2);
}
```

```
/*********************************************************
*                                                        *
*  Name       : Cost                                     *
*  Parameters : n = solution vector                      *
*  Function   : Return the cost of the solution node     *
*               represented by n.                        *
*                                                        *
*********************************************************/
cost(n)
NODE n;
{
     int  m,i,c;

     m = 0;
     for(i=1; i<n.vector[0]; i++)
         if (n.vector[i]== 0)
              m = i;

     c = 0;
     for(i=1; i<=m; i++)
         if (n.vector[i] == 0)
              c += job_list[i].p;

     return(c);
}


/*********************************************************
*                                                        *
*  Name       : Bound                                    *
*  Parameters : n = solution vector                      *
*  Function   : Return the upper bound of node n         *
*                                                        *
*********************************************************/
bound(n)
NODE n;
{
     int i,b;

     b = 0;
     for(i=1; i<=number_of_jobs; i++)
         if (n.vector[i] != 1)
              b += job_list[i].p;

     return(b);
}
```

```
/**********************************************************
*                                                         *
*  Name       : Max                                       *
*  Parameters : Last = Last Job placed in job schedule    *
*  Function   : Return the maximum deadline of those      *
*               jobs included in the schedule so far.     *
*                                                         *
**********************************************************/
max(last)
int last;
{
     int i,
     m;

     m = job_list[last].d;
     for(i=1; i<=last; i++)
         if ((E_node.vector[i] == 1) &&
              (m < job_list[i].d))
               m = job_list[i].d;

     return(m);
}



/**********************************************************
*                                                         *
*  Name       : Create Init Jobs                          *
*  Parameters : none                                      *
*  Function   : Create the initial set of problems to     *
*               solve.  This translates to building the   *
*               first levels of the search tree.  Use a   *
*               least-cost branch and bound search        *
*               technique while creating these initial    *
*               problems.                                 *
*                                                         *
**********************************************************/
create_init_jobs()
{
NODE E_node;                                   /* Solution Vector        */

int  i,                                        /* Iteration Counter      */
     best_penalty,                             /* Penalty of Best Answer */
     deadline,                                 /* Deadline Constraint    */
     job,                                      /* Schedule this job      */
     time_bound,                               /* Time Constraint        */
     new_bound,                                /* Penalty of new vector  */
     new_bound;                                /* Bound of new vector    */
```

```
/*----- Initialize the problem queue with a solution vector and set
        the cost of the best answer to infinity -----*/

      E_node.vector[0] = 1;
      E_node.cost      = u_bound;

      for(i=1; i<=number_of_jobs; i++)
          E_node.vector[i] = 9;

      insert_priority(E_node);

      best.cost = INFINITY;

/*----- Add problems to the queue until there are 4 times the number of
        nodes in the hypercube of problems  or  until the queue is empty
        - If the queue goes empty, then the problem was considered trival
          and the control process solved it sequentially -----*/

      while (( q_length < (number_of_nodes*4)) && (q_status != EMPTY)){
          E_node = delete_q();

      /*----- Get the E-node from the queue and calculate its Deadline
          and Total Run time ----*/

          job = E_node.vector[0];
          if (job <= number_of_jobs){
              E_node.vector[0]++;

      /*----- Generate the first child (add the job to the schedule)
                - Calculate the Deadline and the Total Time Constraints
                - Add the next job if the deadline is greater than or
                  or equal to the time constraint -----*/

              if (job <= number_of_jobs){
                  time_bound = 0;
                  deadline   = max(job);

                  for(i=1; i<job; i++)
                      if (E_node.vector[i] == 1)
                          time_bound += job_list[i].t;
                  time_bound += job_list[job].t;

                  if (deadline >= time_bound){
                      E_node.vector[job] = 1;
                      E_node.cost = cost(E_node);
                      if (E_node.cost < u_bound){
                          if ((new_bound = bound(E_node)) < best_bound)){
                                best = copy_node(E_node,best);
                                best_bound = bound(best);
                          }
```

```
                        insert_priority(E_node);
                    }

                    if (new_bound < u_bound) u_bound = new_bound;
                }

        /*----- Generate the second child (don't schedule the job)
                - If the upper bound of the child is less than the
                  present upper bound, then a new best answer has been
                  found and a new upper bound has been found -----*/

                  E_node.vector[job] = 0;
                  E_node.cost = cost(E_node);
                  if (E_node.cost < u_bound){
                      if ((new_bound  = bound(E_node)) < best_bound)){
                           best = copy_node(E_node,best);
                           best_bound = bound(best);
                      }
                      insert_priority(E_node);
                  }

                  if (new_bound < u_bound) u_bound = new_bound;
              }
          }
      }
  }
```

The third program for the parallel deadline job scheduling search is the worker process. It is defined by the C Language source code on the next few pages. The worker process does not resemble its sequential counterpart. For the sequential deadline job scheduling solution, the entire search is conducted using the least-cost branch and bound technique. Because of memory problems, only the control process can keep the initial problems in least-cost order. The worker processes search the subtrees (problems) using a simple depth-first search. With this information, the worker process begins its initialization once loaded by the intermediate host. The initial steps include receiving the number of jobs to schedule and the 3-tuple for each job. The last initialization step is to send a 'work request' to the control process. Next, the worker process enters an infinite control loop with two operations. The first operation monitors for an E-node message from the control process. Upon receiving the E-node, find the best answer in the subtree. Once the subtree is investigated, send a 'work request' to the control process. The second operation is to monitor for a KILL message. Upon receiving a KILL message, terminate the search.

```
/*********************************************************************
*                              THESIS                                *
*                                                                    *
*                  iPSC Deadline Job Scheduling                      *
*                  iPSC Worker Process Program                       *
*                                                                    *
*                                                                    *
*  Date:  5 Nov 1986                                                 *
*                                                                    *
*  Functions:  This is the iPSC Worker Process program for the       *
*         parallel deadline job scheduling problem.  This            *
*         program needs the following information from the           *
*         intermediate host for initialization:                      *
*                                                                    *
*         1- Number of Jobs to Schedule                              *
*         2- (Penalty, Deadline, Time) 3-tuple defined for           *
*             each of the jobs                                       *
*         3- The initial upper bound                                 *
*                                                                    *
*         After receiving this information, the worker process       *
*         enters its control loop.  Inside this two operations       *
*         are performed, (1) Solve a problem for the Control         *
*         Process, and (2) Terminate the search upon receiving       *
*         a KILL message.                                            *
*                                                                    *
*  Libraries: Standard Input & Output                                *
*             iPSC cnode definitions                                 *
*                                                                    *
*  Language:  C Language                                             *
*                                                                    *
*  Operating System: iPSC Release 2.1                                *
*                                                                    *
*  Message from Host: Number of Jobs to Schedule                     *
*                     (Penalty, Deadline, Time) Job ID 3-tuple       *
*                      Initial Upper Bound                           *
*                                                                    *
*  Message to Host:  None                                            *
*                                                                    *
*  Message from Control: Solve Subtree (E_node)                      *
*                           Termination (Kill) Message               *
*                                                                    *
*  Message to Control: Work Request                                  *
*                                                                    *
*  Author: Capt Rick Mraz                                            *
*                                                                    *
*********************************************************************/
```

```
/************************************************
*               Header Files                    *
************************************************/
#include <stdio.h>                      /* Standard IO              */
#include "/usr/ipsc/lib/cnode.def"      /* Hypercube Node Header    */
#include "q.h"                          /* Solution Vector          */
#include "job.h"                        /* Define Job 3-tuple       */


/************************************************
*               Define Constants                *
************************************************/
#define HOST            0x8000          /* Intermediate Host        */
#define   NODE_PID      0               /* Node Process ID          */
#define CONTROLLER      0               /* Controller = Node 0      */
#define   HOST_PID      1               /* Host Process ID          */
#define INFINITY        9999            /* Need a big number ??     */

#define E_NODE          10              /* Generate Children Msg    */
#define WORK_REQUEST    40              /* Worker needs work...     */
#define NUM_JOBS        50              /* Rec Num Jobs to schedule */
#define NEW_JOB         60              /* Rec a Job 3-tuple        */
#define U_BOUND         70              /* Rec Initial Upper Bound  */
#define KILL            80              /* Kill worker Process      */
#define BEST            200             /* Send Local Best to Control*/


/************************************************
*           Define Global Variables             *
************************************************/
int     cid,                            /* Channel ID               */
        child_count,                    /* Work Load Count          */
        u_bound,                        /* Upper Bound              */
        number_of_jobs;                 /* Number of Jobs to Schedule*/

JOB     job_list[MAX_JOBS];             /* Array of Job 3-tuples    */

NODE    E_node,                         /* Next node to expand      */
        best,                           /* Local Best Answer        */
        copy_node();                    /* Copy one vector to another*/
```

```
/*********************************************************
*                     Main Program                       *
*********************************************************/
main()
{

int     i,                                  /* Iteration Counter         */
        cnt,                                /* Msg Length                */
        ignore,                             /* Forget this parameter     */
        me,                                 /* My node number            */
        node,                               /* Node number               */
        pid;                                /* Process ID                */

/*----- Initialize the Control Process by....
        - Open a communications channel
        - Receive the Number of Jobs to Schedule
        - Receive the initial Upper Bound
        - Receive the Job Triple information for all jobs -----*/

        cid = copen(NODE_PID);

        recvw(cid,NUM_JOBS,&number_of_jobs,sizeof(number_of_jobs),
        &cnt,&node,&pid);

        recvw(cid,U_BOUND,&u_bound,sizeof(u_bound),&cnt,&node,&pid);

        for(i=1; i<=number_of_jobs; i++)
           recvw(cid,NEW_JOB,&job_list[i],sizeof(JOB),&cnt,&node,&pid);

        sendw(cid,WORK_REQUEST,&ignore,sizeof(ignore),
              CONTROLLER,NODE_PID);

        best.cost = INFINITY;

/*----  Enter the control loop
        - If an Enode msg arrives, solve the subtree denoted by the enode.
          After solving the subtree, request more work.
        - If a KILL msg arrives, terminate the control loop -----*/

        for(;;){
           if (probe(cid,E_NODE) >= 0){
                recvw(cid,E_NODE,&E_node,sizeof(NODE),&cnt,&node,&pid);
                solve_subtree(E_node);
                sendw(cid,WORK_REQUEST,&ignore,sizeof(ignore),
                      CONTROLLER,NODE_PID);
           }
           if(probe(cid,KILL)>=0) break;
        }
        cclose(cid);
}
```

```
/**********************************************************
*                                                         *
*  Name     : Cost                                        *
*  Parameters: n = solution vector                        *
*  Function: Return the cost of the solution node         *
*            represented by n.                            *
*                                                         *
**********************************************************/
cost(n)
NODE n;
{
       int  m,i,cost;

       m = 0;
       for(i=1; i<n.vector[0]; i++)
           if ((n.vector[i]== 0))
                       m = i;

       cost = 0;
       for(i=1; i<=m; i++)
           if (n.vector[i] == 0)
                       cost += job_list[i].p;

       return(cost);
}


/**********************************************************
*                                                         *
*  Name     : Max                                         *
*  Parameters: Last = Last Job Scheduled                  *
*  Function: Return the maximum deadline of those         *
*            jobs included in the schedule so far.        *
*                                                         *
**********************************************************/
max(E_node,last)
NODE E_node;
int last;
{
       int i,m;

       m = job_list[last].d;
       for(i=1; i<=last; i++)
           if ((E_node.vector[i] == 1) &&
               (m < job_list[i].d))
                       m = job_list[i].d;

       return(m);
}
```

```
/************************************************************
*                                                           *
*  Name    : Max                                            *
*  Parameters: Last = Last Job Scheduled                    *
*  Function: Return the maximum deadline of those           *
*            jobs included in the schedule so far.          *
*                                                           *
************************************************************/
get_best_bound(u)
int u;
{
int     node,
        pid,
        cnt,
        new_bound;

        while (probe(cid,U_BOUND) >= 0){
           recvw(cid,U_BOUND,&new_bound,sizeof(new_bound),
                 &cnt,&node,&pid);
           if (new_bound < u) u = new_bound;
        }
        return(u);
}

/************************************************************
*                                                           *
*  Name    : Copy Node                                      *
*  Parameters: n1 = Source Node                             *
*              n2 = Destination Node                        *
*                                                           *
*  Function: Copy node, n1, into node, n2,  and return      *
*            node, n2.                                      *
*                                                           *
************************************************************/
NODE copy_node(n1,n2)
NODE n1,n2;
{
        int i;

        for(i=0; i<=VECTOR_SIZE; i++)
           n2.vector[i] = n1.vector[i];

        n2.cost = n1.cost;
        return(n2);
}
```

```
/************************************************************
*                                                           *
*  Name     : Bound                                         *
*  Parameters: n = solution vector                          *
*  Function: Return the upper bound of node n               *
*                                                           *
************************************************************/
bound(n)
NODE n;
{
int     i,
        p;

        p = 0;
        for(i=1; i<=number_of_jobs; i++)
            if (n.vector[i] != 1)
                        p += job_list[i].p;

        return(p);
}



/************************************************************
*                                                           *
*  Name     : Solve Subtree                                 *
*  Parameters: E-node = Subtree (problem) to search         *
*  Function: Given the subtree defined by the  E_node,      *
*            search the subtree for an answer using a       *
*            depth-first search technique.                  *
*                                                           *
*            If a new 'local' best answer was found         *
*            send it to the Control Process before          *
*            returning.                                     *
*                                                           *
************************************************************/
solve_subtree(E_node)
NODE E_node;
{

int     job,                    /* Schedule this job                 */
        i,                      /* Iteration counter                 */
        new_best_flag,          /* Found a new best?                 */
        root,                   /* Root of subtree                   */
        deadline,               /* Schedule Deadline                 */
        time_bound,             /* Time Constraint                   */
        best_bound,             /* Penalty of best node              */
        new_bound;              /* new upper bound                   */
```

```
job  = E_node.vector[0];
root = job - 1;
new_best_flag = FALSE;

/*----- Only solve the subtree defined by the root -----*/
while (job > root){

   switch (E_node.vector[job]){

/*--- Case 9: Add the Job to the schedule ---*/
   case 9:
              deadline   = max(E_node,job);

              time_bound = 0;
              for(i=1; i<job; i++)
                   if (E_node.vector[i] == 1)
                      time_bound += job_list[i].t;
              time_bound += job_list[job].t;

              E_node.vector[job] = 1;
              if (deadline >= time_bound){
                   E_node.cost = cost(E_node);
                   u_bound = get_best_bound(u_bound);
                   if (E_node.cost < u_bound)
                      if ((new_bound = bound(E_node)) <
                          best_bound){
                          new_best_flag = TRUE;
                          best = copy_node(E_node,best);
                          best_bound = bound(E_node);
                      }

                   if (new_bound < u_bound) u_bound = new_bound;

                   if (job < number_of_jobs)  job++;
              }

              break;

/*--- Case1: Do not schedule the next job ---*/
   case 1:
              E_node.vector[job] = 0;
              E_node.cost = cost(E_node);

              if (E_node.cost < u_bound){
                   u_bound = get_best_bound(u_bound);
                   if ((new_penalty = bound(E_node)) <
                        best_penalty){
                        best = copy_node(E_node,best);
                        new_best_flag = TRUE;
                        best_bound = bound(E_node);
                   }
```

```
                              if (new_bound < u_bound) u_bound = new_bound;

            /*--- If the present job is less than the total number of jobs
                  to schedule, then
                  - Schedule the next job by incrementing 'job'
                  - Otherwise, this is the terminal node of the left-most
                    branch of the search tree.....therefore, continue the
                    depth-first search by backtracking to the right-branch
                   (decrement 'job') ---*/

                              if (job < number_of_jobs)
                                 job++;
                              else{
                                 E_node.vector[job] = 9;
                                 job--;
                              }
                         }

                         break;

      /*--- Case 0:  Reached the terminal node of the right-most
                     branch of the tree...therefore, backtrack
                     by decrementing 'job' ---*/
         case 0:
                     E_node.vector[job] = 9;
                     job--;
                     break;
         }
      }

      /*---- If a new local best is found,
              send it to the Control Process -----*/
      if (new_best_flag)
         sendw(cid,BEST,&best,sizeof(NODE),CONTROLLER,NODE_PID);
}
```

Appendix D:

Sequential Deadline Job Scheduling Source Code Description

The listing on the following pages was run on a DEC VAX 11/785. The run times for various board sizes sets a baseline for performance comparisons with the iPSC Hypercube (see Chapter VI, Performance Measures and Experiment Results). This version of deadline job scheduling is written in C Language. The description of the code follows.

Pages 1 and 2 of the listing define the Header Files, Constants, and Variables used in the program. The main program also starts on the second page of the listing. First, the initial state of the problem is set. The E_node is initialized, and the upper_bound is set to INFINITY, and the 'best' solution is set. Next, the number of jobs is read from standard input followed by the initialization of the job_list array. Once the jobs are read, the initial problem vector, $(x_1,x_2,\ldots,x_n)$ where n = the number of jobs to be scheduled is set to $(1,\bullet,\ldots,\bullet)$ where • defines an x_i that has not been determined. The number 1 in the first element, vector[0], identifies job #1 is the first job to try to schedule. Then, the cost of the initial problem is set equal to the upper_bound. The live node queue is then initialized with this first vector.

Next, the while loop is executed until the first solution is found. This condition occurs once all live nodes have been examined but, while there is something in the queue, the following procedure is conducted. First, get an E-node (the next node to expand) from the front of the live node queue. The next job to schedule for that particular E-node can be found in the first element of the vector, vector[0].

Before generating children, increment the first element, vector[0]. This sets the state of the solution vector such that the next job is scheduled if this child becomes a live node. Then, determine the maximum deadline and the total run time of those jobs scheduled so far. This information is needed to generate the first child.

Now, generate the first child. This child attempts to add the next job to the schedule. The next job can be scheduled only if the maximum deadline is greater than or equal to the total run time plus the time to run the next job. If this job meets these requirements, schedule it by setting the value of its vector to 1, and compute the cost of the child. Next, the child becomes a live node (inserted into the live node queue) only if the cost of the child is less than the upper bound. If the child is a live node, it may also be the best solution so far. The live node is best solution if the cost of the live node is less than the upper_bound and the total penalty of the live node is less than the total penalty of the best solution. Finally, before leaving the first child generation, check to see if the child sets a new upper_bound.

Now it's time to generate the second child. This child does not have to pass the time and deadline test since it does not add this job to the schedule. First, do not schedule this job by setting its vector to 0. Next, calculate the cost of the child. If the cost is less than the upper bound, then is becomes a live node (insert it into the live node queue). If this child becomes a live node, also check for best solution so far. The conditions for the live node and the best solution are the same for this child as they were for the first child. Finally, check to see if the child sets a new upper_bound.

Once the solution has been found, take a stop time and print the best job schedule as well as the time to find that schedule.

```
/**********************************************************************
*                              THESIS                                 *
*                                                                     *
*                 Sequential Deadline Job Scheduling                  *
*                                                                     *
*    Date : 27 Aug 1986                                               *
*    Function: Given a set of job described with the following       *
*              parameters,                                            *
*                                                                     *
*                    - Deadline to finish running by                  *
*                    - Penalty if the job does not run                *
*                    - Time to run the job                            *
*                                                                     *
*              Find the largest subset of jobs that can be run *
*              by their deadline as well as minimize the total *
*              penalty paid.                                          *
*                                                                     *
*    Language: C Language                                             *
*                                                                     *
*    Operating System: 4.2 BSD UNIX                                   *
*                                                                     *
*    Libraries: Standard I/O     Types.h      Times.h                 *
*                                                                     *
*    Author: Captain Rick Mraz                                        *
*                                                                     *
**********************************************************************/
/***********************************************
*                HEADER FILES                  *
***********************************************/
#include <stdio.h>                      /* Standard IO              */
#include </sys/h/types.h>               /* Time structure type      */
#include </sys/h/times.h>               /* Time functions           */
#include "q.h"                          /* Queue Routine Header     */
#include "j.h"                          /* Job information Header   */


/***********************************************
*                  CONSTANTS                   *
***********************************************/
#define INFINITY 999999                 /* Need a large number??    */
```

```
/***********************************************
*              VARIABLES                      *
***********************************************/
int  i,                             /* Iteration Counter          */
     best_penalty,                  /* Penalty of best solution */
     deadline,                      /* Maximum Deadline           */
     new_penalty,                   /* Penalty of live node       */
     number_of_jobs,                /* Total number of jobs       */
     job,                           /* Schedule this job          */
     time_bound,                    /* Total run time of jobs     */
     new_bound,                     /* new upper bound            */
     upper_bound;                   /* Upper Bound                */

NODE   live_node,                   /* Live node                  */
       best,                        /* Best node so far...        */
       E_node,                      /* Next node to expand        */
       copy_node(),                 /* Clone a Node...            */
       temp;                        /* Temporary Vector           */


JOB    job_list[MAX_JOBS];          /* Array of Jobs to Schedule*/

struct tms start_time;              /* Start of Computation       */
struct tms stop_time;               /* End of Computation         */


/***********************************************
*      Main Program...Sequential              *
*      Deadline Job Scheduling                *
***********************************************/
main() {

/*----- Initialize the job scheduler ----- */
       times(&start_time);

       upper_bound = INFINITY;
       best.cost = INFINITY;
       q_init();

       /*----- Read in the list of jobs to schedule ----- */
       scanf("%d",&number_of_jobs);
       for(i=1; i<=number_of_jobs; i++)
            scanf("%d%d%d",&job_list[i].p,&job_list[i].d,&job_list[i].t);
```

```
        /*----- Initialize the Problem Vector & Live Node Q ----- */
        temp.vector[0] = 1;
        for(i=1; i<=number_of_jobs; i++)
            temp.vector[i] = 9;

        temp.cost = upper_bound;
        insert_priority(temp);

/*----- Find the best schedule -----*/
        while (q_status != EMPTY){
           delete_q(E_node,q);                    /* Get E-node from queue     */

           job = E_node.vector[0];                /* Schedule this job         */
           if (job <= number_of_jobs){
              E_node.vector[0]++;                 /* Schedule the next job...  */

              if (job <= number_of_jobs){         /* Calculate Time & Deadline*/
                 time_bound = 0;
                 deadline = max(E_node,job);

                 for(i=1; i<=job; i++)
                    if (E_node.vector[i] == 1)
                       time_bound += job_list[i].t;

        /*----- Generate First Child ----- */
                 if (deadline >= (time_bound + job_list[job].t)){
                    E_node.vector[job] = 1;
                    E_node.cost = cost(E_node);

                    if (E_node.cost < upper_bound){

                       live_node = copy_node(E_node,live_node);
                       insert_priority(live_node);

                       if ((new_penalty = total_penalty) <
                           (best_penalty = total_penalty))
                                 best = copy_node(live_node,best);

                    }
                    if ((new_bound = bound(E_node)) < upper_bound)
                        upper_bound = new_bound;
                 }
```

```
        /*----- Generate Second Child ----- */
                E_node.vector[job] = 0;
                E_node.cost = cost(E_node);

                if (E_node.cost < upper_bound){
                   insert_priority(E_node);

                   if ((new_penalty = total_penalty) <
                       (best_penalty = total_penalty))
                              best = copy_node(E_node,best);

                }

                if ((new_bound = bound(E_node)) < upper_bound)
                     upper_bound = new_bound;
             }
          }
       }

/*----- Computation Complete...Take Time Measure -----*/
       times(&stop_time);

       print_solution(best);
       printf("\nTime to Solution: %f sec\n",
          (float)(stop_time.tms_utime-start_time.tms_utime)/60.0);
}

/*******************************************************
*                                                      *
*   Name: Bound                                        *
*   Parameters: n = calculate the bound of this node   *
*                                                      *
*   Function: Calculate the bound of the node, n.  The *
*               bound of a node equals the sum of all  *
*               penalties of those jobs not included in *
*               the job schedule so far.               *
*                                                      *
* *****************************************************/
bound(n)
NODE *n;
{
int     i,
        b;

    b = 0;
    for(i=1; i<=number_of_jobs; i++)
        if ((n->vector[i] == 0) || (n->vector[i] == 9))
             b += job_list[i].p;

    return(b);
}
```

```
/**********************************************************
*                                                         *
*  Name: Copy Node                                        *
*  Parameters: n = Copy this node                         *
*  Function: Copy the Node, n, into a new node and        *
*                return a pointer to the new node.        *
*                                                         *
**********************************************************/
NODE copy_node(n1,n2)
NODE n1,n2;
{
int     i;

   for (i=0; i<=VECTOR_SIZE; i++)
      n2.vector[i] = n1.vector[i];
      n2.cost = n1.cost;

   return(n2);
}

/**********************************************************
*                                                         *
*  Name: Max                                              *
*  Parameters: n = Use this vector                        *
*                   job = find max deadline compared to   *
*                          this job                       *
*                                                         *
*  Function: Find the maximum deadline of the vector      *
*               n and the job                             *
*                                                         *
**********************************************************/
max(n,job)
NODE n;
int  job;
{
int    i,
       m;

   m = job_list[job].d;
   for(i=1; i<=last; i++)
      if ((E_node.vector[i] == 1) && (m < job_list[i].d))
         m = job_list[i].d;

   return(m);
}
```

```
/*********************************************************
*                                                        *
*  Name: Cost                                            *
*  Parameters: n = Calculate the cost of this node       *
*                                                        *
*  Function: The cost of the job scheduling vector       *
*                an additional parameter, 'm'.  m is     *
*               defined as follows,                      *
*                                                        *
*                     m = max(Sx)                        *
*                                                        *
*                     where Sx is the set of jobs        *
*                     examined so far.                   *
*                                                        *
*               The cost is then computed using the      *
*               following summation,                     *
*                                                        *
*                        Σ   pi                          *
*                        i<m                             *
*                        i∉J                             *
*                                                        *
*                     where J is the set of jobs scheduled *
*                     so far and pi is the penalty for   *
*                     job i.                             *
*                                                        *
*********************************************************/
cost(n)
NODE n;
{
int m,
    i,
    c;

    m = 0;
    for(i=1; i<n.vector[0]; i++)
        if (n.vector[i] == 0)
            m = i;

    c = 0;
    for(i=1; i<=m; i++)
        if (n.vector[i] == 0)
            c += job_list[i].p;

    return(c);
}
```

```
/*********************************************************
*                                                       *
*   Name: Print Solution                                *
*   Parameters: b = best solution                       *
*                                                       *
*   Function: Print the best job schedule given the best *
*             vector.  Also print the total penalty paid *
*                                                       *
*********************************************************/
print_solution(b)
NODE b;
{
int    i,
       penalty;

    printf("The Best Job Schedule is:\n");
    printf("-----------------------\n");
    printf("\tJob(s): ");

    penalty = 0;
    for(i=1; i<=number_of_jobs; i++){
        if (best.vector[i] == 1)
            printf("%d ",i);

        if (best.vector[i] != 1)
         penalty += job_list[i].p;
    }

    printf("\n\n");
    printf("\nPenalty = %d\n",penalty); print_vector(best);
}


/*********************************************************
*                                                       *
*   Name: Total Penalty                                 *
*   Parameters: n = Find the Total Penalty of this job  *
*                                                       *
*   Function: Return the total penalty paid.            *
*                                                       *
*********************************************************/
total_penalty(n)
NODE *n;
{
    int  i,p;

    for(i=1; i<=number_of_jobs; i++)
        if (n.vector[i] != 1)
            p += job_list[i].p;
    return(p);
}
```

```
/*******************************************************************
*                    Job Information Header File                  *
*                                                                  *
*              Sequential Deadline Job Scheduling                 *
*                                                                  *
*    Date:  28 Aug 1986                                           *
*                                                                  *
*    Author:  Captain Rick Mraz                                   *
*                                                                  *
*    Language: C Language                                         *
*                                                                  *
*                                                                  *
*******************************************************************/
/**********************************************
*    Header files                             *
**********************************************/
#include <stdio.h>                            /* Standard I/O          */


/**********************************************
*    Constants                                *
**********************************************/
#define MAX_JOBS    30                        /* Maximum of 30 jobs    */


/**********************************************
*    Node Information Record                  *
**********************************************/
struct job {
        int p;                                /* Penalty               */
        int d;                                /* Deadline              */
        int t};                               /* Time to Run           */

typedef struct job JOB;                       /* Define the type JOB   */

/***************************************************
*    FUNCTION DEFINITIONS- job queue functions     *
***************************************************/
void insert_priority();
void deleteq();
```

To re-enforce the least-cost branch & bound algorithm, an example problem is in order. Given the following job sequence (16:384),

Job	p_i	d_i	t_i
1	5	1	1
2	10	3	2
3	6	2	1
4	3	1	1

find the subset of jobs such that the penalty paid is minimal. The solution vector for this problem, (x_1,x_2,x_3,x_4), has the following explicit constraints, $x_i \in \{1,0\}$ where 1 denotes that job i is included in the schedule and 0 denotes that job i has not been included in the schedule. For example, solution vector (1,1,1,1) identifies all jobs have been scheduled. The entire solution space of this problem is shown in Figure D1. Next, the implicit constraints define the relationships between the x_i's. The first implicit constraint, Deadline/Total Time constraint, is quite easy to understand, the next job can be scheduled only when the maximum deadline of the jobs under consideration is greater than or equal to the total run time of those same jobs. For example, the grey node in Figure D1 identifies the solution vector (1,1,1,•) that does not pass the implicit constraint because the maximum deadline of jobs 1,2,and 3 equals 3 and the total run time of those same jobs equals 4. The second implicit constraint, Cost/Upper Bound constraint, relies upon the cost of the node and the value of the upper bound. The number inside each node of Figure D1 identifies the cost of that node. See Chapter 5 for the details on how the cost and upper bounds are calculated.

Figure 37: Example 4-Job Deadline Job Scheduling Solution Space

Combining the explicit constraints and the implicit constraints the search tree to find the answer to this job set looks like Figure D2. The square nodes identify those children that can not be scheduled because of the Deadline/Total Run Time constraint. The nodes with the B beside them identify those nodes that failed the Cost/Upper Bound constraint. In the following trace of least-cost branch & bound, the Live Node Q shows the list of live nodes ordered by least-cost (the number to the right is the cost of that vector); the Upper Bound at that point in the search; and the Best Solution (total penalty of that solution is to the right of the vector). With this information, the trace of this least-cost branch & bound search goes as follows:

Live Node Q		U Bound	Best Solution		Remarks
(•,•,•,•)	0	24	(0,0,0,0)	24	Initial Problem
-- Expand next E-node (•,•,•,•)					
(1,•,•,•)	0	19	(1,•,•,•)	19	1st Child, update u_bound & best
(1,•,•,•) (0,•,•,•)	0 5	14	(1,•,•,•)	19	2nd Child, update u_bound
-- Expand next E-node (1,•,•,•)					
(1,1,•,•) (0,•,•,•)	0 5	9	(1,1,•,•)	9	1st Child, update u_bound & best
(1,1,•,•) (0,•,•,•) (1,0,•,•)	0 5 10	9	(1,1,•,•)	9	2nd Child
-- Expand next E-node (1,1,•,•)					
(1,1,1,•)	Bound				Deadline/Total Time Constraint
(0,•,•,•) (1,1,0,•) (1,0,•,•)	5 6 10	9	(1,1,•,•)	9	2nd Child

Figure 38: Portion of the 4-Job Solution Space
Generated During Search

Live Node Q		U Bound	Best Solution		Remarks
-- Expand next E-node (0,•,•,•)					
(0,1,•,•)	5	9	(1,1,•,•)	9	1st Child
(1,1,0,•)	6				
(1,0,•,•)	10				
(0,1,•,•)	5	9	(1,1,•,•)	9	2nd Child
(1,1,0,•)	6				
(1,0,•,•)	10				
(0,0,•,•)	15				
-- Expand next E-node (0,1,•,•)					
(0,1,1,•)	5	8	(0,1,1,•)	8	1st Child, update u_bound & best
(1,1,0,•)	6				
(1,0,•,•)	10				
(0,0,•,•)	15				
(0,1,1,•)	5	8	(0,1,1,•)	8	2nd Child
(1,1,0,•)	6				
(1,0,•,•)	10				
(0,1,0,•)	11				
(0,0,•,•)	15				
-- Expand next E-node (0,1,1,•)					
(0,1,1,1) Bound					Deadline/Total Time Constraint
(0,1,1,0) Bound					Cost/Upper Bound Constraint
-- The live node queue now looks like this					
(1,1,0,•)	6	8	(0,1,1,•)	8	
(1,0,•,•)	10				
(0,1,0,•)	11				
(0,0,•,•)	15				
-- Expand next E-node (1,1,0,•)					
(1,1,0,1) Bound					Deadline/Total Time Constraint
(1,1,0,0) Bound					Cost/Upper Bound Constraint
-- The live node queue now looks like this					
(1,0,•,•)	10	8	(0,1,1,•)	8	
(0,1,0,•)	11				
(0,0,•,•)	15				

Live Node Q	U Bound	Best Solution	Remarks
-- Expand next E-node (1,0,•,•)			
(1,0,1,•) Bound			Cost/Upper Bound Constraint
(1,0,0,•) Bound			Cost/Upper Bound Constraint
-- The live node queue now looks like this			
(0,1,0,•) 11 (0,0,•,•) 15	8	(0,1,1,•) 8	
-- Expand next E-node (0,1,0,•)			
(0,1,0,1) Bound			Cost/Upper Bound Constraint
(0,1,0,0) Bound			Cost/Upper Bound Constraint
-- The live node queue now looks like this			
(0,0,•,•) 15	8	(0,1,1,•) 8	
-- Expand next E-node (0,0,•,•)			
(0,0,1,•) Bound			Cost/Upper Bound Constraint
(0,0,0,•) Bound			Cost/Upper Bound Constraint

-- The live node queue is now empty and the search terminates with the following job schedule as the best answer:

Job Subset = {2,3}
Penalty = 8

Appendix E:

Tables of Branch and Bound Experiments

The following tables tally the results of the N-queens and the deadline job scheduling experiments. Dash entries could not be calculated because some test were not made. The analysis of this data can be found in Chapter VI, Performance Analysis and Experiment Results. In that chapter, some of this tabular data has been plotted to show trends and for comparisons.

N-queens Experimental Results

Table 1
N-queens VAX Baseline

Board Size	Time to First Solution (sec)	Time to All Solutions(sec)
4	0.0000	0.0000
5	0.0000	0.0167
6	0.0167	0.0734
7	0.0017	0.3848
8	0.0999	2.0042
9	0.0333	10.1958
10	0.1253	54.1718
11	0.0685	308.1500
12	0.4778	1837.5555

Table 2
N-queens Elxsi Computation Time

Board Size	Time to First Solution (sec)	Time to All Solutions(sec)	Speed Up Over VAX
4	0.0000	0.0000	0.000
5	0.0000	0.0167	1.000
6	0.0000	0.0500	1.468
7	0.0000	0.2500	1.539
8	0.0500	1.1667	1.717
9	0.0167	5.9400	1.717
10	0.0667	30.6500	1.767
11	0.0333	172.2500	1.789
12	0.2667	1045.5333	1.758

Table 3
N-queens iPSC d-5 Computation Time

Board Size	Start Up	Time to First Solution (sec)	Time to All Solutions(sec)	Speed Up Over VAX	Speed Up Over d-0
4	0.384	0.448	0.520	0.000	0.154
5	0.416	0.400	0.624	0.027	0.058
6	0.496	0.450	1.024	0.077	0.166
7	0.800	0.764	2.468	0.156	0.314
8	0.544	0.49	2.206	0.908	1.717
9	0.512	0.400	2.198	4.639	8.771
10	0.490	0.400	4.084	13.264	24.823
11	0.496	0.424	18.144	16.984	-----
12	0.464	0.704	110.538	16.624	-----
13	-----	0.656	-----	-----	-----
14	-----	1.328	-----	-----	-----
15	-----	1.776	-----	-----	-----
16	-----	4.240	-----	-----	-----
17	-----	3.248	-----	-----	-----
18	-----	11.744	-----	-----	-----
19	-----	15.920	-----	-----	-----
20	-----	11.248	-----	-----	-----

Table 4
N-queens iPSC d-4 Computation Time

Board Size	Start Up	Time to First Solution (sec)	Time to All Solutions(sec)	Speed Up Over VAX	Speed Up Over d-0
4	0.192	0.368	0.440	0.000	0.018
5	0.208	0.180	0.384	0.044	0.094
6	0.288	0.234	1.086	0.068	0.157
7	0.288	0.266	1.066	0.361	0.726
8	0.256	0.212	1.114	1.799	3.400
9	0.240	0.208	1.824	5.590	10.569
10	0.240	0.434	7.006	7.732	14.470
11	0.224	0.284	39.984	7.707	-----
12	0.240	0.504	270.480	6.794	-----
13	-----	0.784	-----	-----	-----
14	-----	2.976	-----	-----	-----
15	-----	1.136	-----	-----	-----
16	-----	3.536	-----	-----	-----
17	-----	2.880	-----	-----	-----
18	-----	26.976	-----	-----	-----
19	-----	18.048	-----	-----	-----
20	-----	20.016	-----	-----	-----

Table 5
N-queens iPSC d-3 Computation Time

Board Size	Start Up	Time to First Solution (sec)	Time to All Solutions(sec)	Speed Up Over VAX	Speed Up Over d-0
4	0.096	0.096	0.166	0.000	0.048
5	0.096	0.096	0.284	0.059	0.127
6	0.176	0.176	0.584	0.126	0.291
7	0.110	0.110	0.556	0.692	1.392
8	0.112	0.128	0.848	2.363	4.467
9	0.120	0.120	2.974	3.428	6.482
10	0.128	0.176	13.744	3.942	7.376
11	0.128	0.192	75.462	5.038	-----
12	0.112	-----	-----	-----	-----

Table 6
N-queens iPSC d-2 Computation Time

Board Size	Start Up	Time to First Solution (sec)	Time to All Solutions(sec)	Speed Up Over VAX	Speed Up Over d-0
4	0.064	0.042	0.240	0.000	0.068
5	0.080	0.064	0.222	0.075	0.162
6	0.080	0.108	0.310	0.237	0.548
7	0.064	0.056	0.488	0.789	1.586
8	0.064	0.060	1.344	1.491	2.819
9	0.064	0.096	6.130	1.663	3.145
10	0.064	0.206	30.896	1.753	3.281
11	0.064	0.314	173.800	1.773	-----
12	0.064	-----	-----	-----	-----

Table 7
N-queens iPSC d-1 Computation Time

Board Size	Start Up	Time to First Solution (sec)	Time to All Solutions(sec)	Speed Up Over VAX	Speed Up Over d-0
4	0.032	0.042	0.116	0.000	0.069
5	0.032	0.032	0.240	0.696	0.15
6	0.032	0.038	0.352	0.209	0.483
7	0.032	0.032	0.894	0.430	0.866
8	0.032	0.064	3.630	0.552	1.044
9	0.032	0.064	17.622	0.579	1.094
10	0.032	0.288	91.668	0.591	1.106
11	0.032	-----	-----	-----	-----
12	0.032	-----	-----	-----	-----

Table 8
N-queens iPSC d-0 Computation Time

Board Size	Time to First Solution (sec)	Time to All Solutions(sec)	Speed Up Over VAX
4	0.000	0.008	0.000
5	0.000	0.036	0.038
6	0.026	0.170	0.065
7	0.002	0.774	0.089
8	0.204	3.788	0.111
9	0.074	19.278	0.126
10	0.254	101.376	0.101
11	-----	-----	-----
12	-----	-----	-----

Table 9
Load Balance for the 11-queens Problem on a d-5 cube

Node Number	E-nodes Expanded	Node Number	E-nodes Expanded
1	10499	16	10365
2	11141	17	10844
3	10850	18	10511
4	11031	19	10527
5	11092	20	10459
6	11321	21	10760
7	10932	22	10331
8	10807	23	10833
9	10647	24	10179
10	10548	25	10738
11	10506	26	11099
12	10931	27	10600
13	10525	28	10947
14	10125	29	10453
15	10635	30	10570
		31	10153

Table 11

Deadline Job Scheduling iPSC d-5 Computation Time
(Time in Seconds)

Number of Jobs	Start Up Time	Run Time Problem Set #1	Speed Up Over VAX	Start Up Time	Run Time Problem Set#2	Speed Up Over VAX
4	0.868	2.392	0.029	0.894	2.408	0.034
5	0.892	2.800	0.026	0.908	2.800	0.033
6	0.902	3.192	0.021	0.964	3.204	0.410
7	0.920	3.598	0.023	1.062	3.600	0.055
8	0.956	4.396	0.019	1.246	3.996	0.102
9	0.976	4.796	0.017	1.672	4.392	0.210
10	0.968	4.788	0.018	1.150	4.810	0.358
11	0.994	5.186	0.017	1.502	5.188	0.833
12	1.012	5.584	0.018	1.480	5.588	2.270
13	1.028	5.986	0.018	1.502	5.982	7.023
14	1.042	6.384	0.017	1.494	6.380	23.567
15	1.076	6.792	0.018	1.528	9.534	58.130
16	1.084	7.184	0.017	-----	-----	-----
17	1.124	7.660	0.018	-----	-----	-----
18	1.160	8.052	0.017	-----	-----	-----
19	1.164	8.448	0.017	-----	-----	-----
20	1.188	9.138	0.017	-----	-----	-----
21	1.196	9.194	0.019	-----	-----	-----
22	1.222	9.584	0.018	-----	-----	-----
23	1.232	10.150	0.018	-----	-----	-----
24	1.264	10.392	0.019	-----	-----	-----
25	1.288	10.812	0.019	-----	-----	-----

Table 12

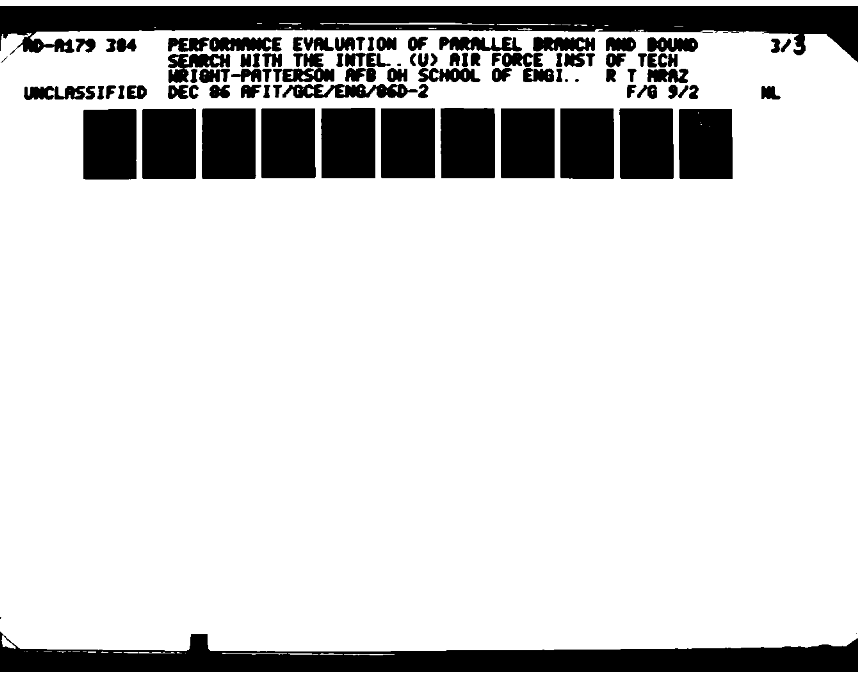
Deadline Job Scheduling iPSC d-4 Computation Time
(Time in Seconds)

Number of Jobs	Start Up Time	Run Time Problem Set #1	Speed Up Over VAX	Start Up Time	Run Time Problem Set#2	Speed Up Over VAX
4	0.468	1.194	0.058	0.482	1.194	0.060
5	0.468	1.400	0.052	0.512	1.702	0.054
6	0.520	1.642	0.041	0.578	1.612	0.081
7	0.526	1.800	0.045	0.660	1.874	0.106
8	0.546	2.006	0.041	0.844	1.990	0.205
9	0.594	2.260	0.040	0.758	2.198	0.420
10	0.580	2.410	0.037	0.764	2.384	0.722
11	0.592	2.598	0.035	0.758	2.592	1.668
12	0.632	2.856	0.036	0.776	2.790	4.546
13	0.630	3.002	0.036	0.820	3.902	10.766
14	0.650	3.196	0.034	0.806	6.792	22.137
15	0.666	3.390	0.035	0.828	12.832	43.190
16	0.690	3.596	0.035	-----	-----	-----
17	0.678	3.790	0.037	-----	-----	-----
18	0.694	4.316	0.032	-----	-----	-----
19	0.732	4.674	0.031	-----	-----	-----
20	0.722	4.444	0.035	-----	-----	-----
21	0.738	4.590	0.038	-----	-----	-----
22	0.748	4.930	0.035	-----	-----	-----
23	0.764	4.994	0.037	-----	-----	-----
24	0.774	5.196	0.037	-----	-----	-----
25	0.790	5.390	0.038	-----	-----	-----

Table 13

Deadline Job Scheduling iPSC d-3 Computation Time
(Time in Seconds)

Number of Jobs	Start Up Time	Run Time Problem Set #1	Speed Up Over VAX	Start Up Time	Run Time Problem Set#2	Speed Up Over VAX
4	0.274	0.594	0.116	0.282	0.594	0.137
5	0.284	0.688	0.106	0.312	0.688	0.133
6	0.308	0.802	0.083	0.368	0.794	0.165
7	0.320	0.894	0.091	0.472	0.914	0.217
8	0.340	0.996	0.082	0.416	0.996	0.410
9	0.342	1.090	0.076	0.418	1.096	0.842
10	0.354	1.190	0.075	0.438	1.312	1.312
11	0.368	1.288	0.071	0.450	1.778	2.431
12	0.382	1.390	0.073	0.456	2.636	4.812
13	0.400	1.494	0.072	0.474	4.324	9.715
14	0.406	1.592	0.068	0.486	7.664	19.619
15	0.420	1.694	0.070	0.506	16.714	33.159
16	0.432	1.794	0.070	-----	-----	-----
17	0.448	1.892	0.074	-----	-----	-----
18	0.464	1.996	0.069	-----	-----	-----
19	0.464	2.094	0.070	-----	-----	-----
20	0.484	2.190	0.069	-----	-----	-----
21	0.502	2.296	0.076	-----	-----	-----
22	0.516	2.532	0.069	-----	-----	-----
23	0.52[illegible]	2.490	0.074	-----	-----	-----
24	0.538	2.588	0.075	-----	-----	-----
25	0.560	2.712	0.075	-----	-----	-----


AD-A179 384
PERFORMANCE EVALUATION OF PARALLEL BRANCH AND BOUND
SEARCH WITH THE INTEL..(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. R T MRAZ
3/3
UNCLASSIFIED
DEC 86 AFIT/GCE/ENG/86D-2
F/G 9/2
NL

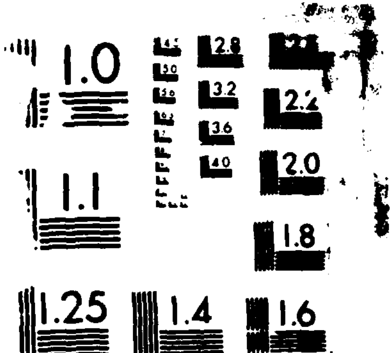


Table 14

Deadline Job Scheduling iPSC d-2 Computation Time
(Time in Seconds)

Number of Jobs	Start Up Time	Run Time Problem Set #1	Speed Up Over VAX	Start Up Time	Run Time Problem Set#2	Speed Up Over VAX
4	0.160	0.290	0.237	0.176	0.294	0.276
5	0.172	0.340	0.214	0.206	0.340	0.270
6	0.192	0.394	0.169	0.258	0.386	0.340
7	0.206	0.440	0.185	0.240	0.576	0.344
8	0.216	0.520	0.156	0.262	0.710	0.575
9	0.226	0.540	0.154	0.270	0.920	1.003
10	0.242	0.594	0.151	0.278	1.318	1.306
11	0.258	0.646	0.142	0.298	2.244	1.926
12	0.268	0.690	0.148	0.322	7.058	1.797
13	0.282	0.740	0.146	0.340	13.836	3.036
14	0.294	0.790	0.137	-----	-----	-----
15	0.316	0.846	0.140	-----	-----	-----
16	0.326	0.900	0.139	-----	-----	-----
17	0.336	0.942	0.148	-----	-----	-----
18	0.344	0.990	0.139	-----	-----	-----
19	0.356	1.044	0.140	-----	-----	-----
20	0.372	1.090	0.141	-----	-----	-----
21	0.384	1.140	0.154	-----	-----	-----
22	0.394	1.192	0.147	-----	-----	-----
23	0.414	1.246	0.149	-----	-----	-----
24	0.428	1.300	0.149	-----	-----	-----
25	0.432	1.336	0.153	-----	-----	-----

Table 15

Deadline Job Scheduling iPSC d-1 Computation Time
(Time in Seconds)

Number of Jobs	Start Up Time	Run Time Problem Set #1	Speed Up Over VAX	Start Up Time	Run Time Problem Set#2	Speed Up Over VAX
4	0.100	0.208	0.331	0.120	0.146	0.556
5	0.112	0.224	0.326	0.130	0.336	0.275
6	0.118	0.246	0.271	0.150	0.434	0.302
7	0.134	0.276	0.294	0.152	0.588	0.336
8	0.146	0.304	0.267	0.168	0.896	0.456
9	0.174	0.342	0.244	0.178	1.446	0.640
10	0.176	0.368	0.243	0.198	2.606	0.660
11	0.194	0.394	0.233	0.212	5.036	0.858
12	0.206	0.420	0.243	0.228	9.812	1.293
13	0.222	0.456	0.238	0.240	13.264	3.167
14	0.230	0.482	0.225	-----	-----	-----
15	0.248	0.512	0.232	-----	-----	-----
16	0.256	0.542	0.231	-----	-----	-----
17	0.272	0.566	0.247	-----	-----	-----
18	0.288	0.598	0.230	-----	-----	-----
19	0.300	0.632	0.231	-----	-----	-----
20	0.314	0.656	0.232	-----	-----	-----
21	0.328	0.696	0.251	-----	-----	-----
22	0.336	0.720	0.243	-----	-----	-----
23	0.350	0.756	0.245	-----	-----	-----
24	0.362	0.784	0.247	-----	-----	-----
25	0.380	0.816	0.250	-----	-----	-----

Table 16
Load Balance - Deadline Job Scheduling
Problem Set #1
20-Jobs Solved on an iPSC d-4

Node Number	E-nodes Expanded	Node Number	E-nodes Expanded
1	125	9	0
2	60	10	0
3	0	11	0
4	0	12	0
5	0	13	0
6	0	14	0
7	0	15	0

Table 17
Load Balance - Deadline Job Scheduling
Problem Set #2
15-Jobs Solved on an iPSC d-4

Node Number	E-nodes Expanded	Node Number	E-nodes Expanded
1	3326	9	2843
2	2806	10	3183
3	6065	11	1959
4	2694	12	2047
5	2765	13	3313
6	3102	14	1984
7	2412	15	1536

Bibliography

1. Akl, Selim G. et al. "Design, Analysis, and Implementation of a Parallel Tree Search Algorithm," IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-4: 192-203 (March 1982).

2. Barr, Avron, and Edward A. Feigenbaum. The Handbook of Artificial Intelligence, Vol 1. Stanford, California: HeurisTech Press, 1981.

3. Booch, Grady. Arpanet Electronic Mail Message, 4 June 1986.

4. Booch, Grady. "Object-Oriented Development," IEEE Transactions on Software Engineering, SE-12: 211-221.

5. Booch, Grady. Software Engineering with Ada. Menlo Park, California: Benjamin/Cummings, 1983.

6. Bosma, John T. and Richard C. Wheelan. Guide to the Strategic Defense Initiative. Arlington, Virginia: Pasha Publications, 1985.

7. Defense Advanced Research Projects Agency (DARPA). Strategic Computing. AD-A141 282/9, Arlington, VA, 1983.

8. Fox, Geoffrey C., and Steve W. Otto. "Algorithms for Concurrent Processors," Physics Today: 13-20 (May 1984).

9. Frenkel, Karen A. "Evaluating Two Massively Parallel Machines," Communications of the ACM: 29 752-758 (August 1986).

10. Fuller, Samuel H. "Performance Evaluation," Introduction to Computer Architecture (Second Edition), edited by Harold S. Stone. Chicago: Science Research Associates, 1980.

11. Gajski, Daniel d., and Jih-Kwon Peir. "Essential Issues in Multiprocessor Systems." Computer: 9-27 (June 1985).

12. Gane, Chris and Trish Sarson. Structured Systems Analysis: tools & techniques (second edition). New York: Improved System Technologies, Inc., 1977.

13. Handler, Wolfgang. "Innovative Computer Architecture-How to increase parallelism but not complexity," Parallel Processing Systems, edited by David J. Evans. Cambridge, MA: Cambridge University Press, 1982.

14. Hillis, W. Daniel. The Connection Machine. Cambridge, Mass: MIT Press, 1985

15. Horowitz, Ellis, and Sartij Sahni. Fundamentals of Data Structures in PASCAL. Rockville, Maryland: Computer Science Press, 1984.

16. Horowitz, Ellis, and Sartij Sahni. Fundamentals of Computer Algorithms. Rockville, Maryland: Computer Science Press, 1978.

17. Intel iPSC Concurrent Programming Workshop Notes, Intel Scientific Computers, Beaverton, Oregon. (16-20 June 1986).

18. Kleinrock, Leonard. "Distributed Systems," Communications of the ACM, 28: 1200-1213 (November 1985).

19. Lee, Lieutenant Ronald. Performance Comparison and Analysis of State of the Art Machines. MS Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1985.

20. Lipschutz, Seymore. Schaum's Outline Series Theory and Problems of Discrete Mathematics. New York: McGraw-Hill, 1976.

21. Norman, Captain Douglas O. Reasoning in Real-Time for the Pilot Associate: An Examination of a Model Based Approach to Reasoning in Real-Time for Artificial Intelligence Systems using a Distributed Architecture. MS Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1985.

22. Patton, Peter C. "Multiprocessors: Architecture and Applications," Computer: 29-40 (June 1985).

23. Page-Jones, Meilir. The Practical Guide to Structured Systems Design. New York: Yourdan Press, 1980.

24. Retelle, LtCol John P. Jr. "The Pilot's Associate-Aerospace Application of Artificial Intelligence," Signal: 100-105 (June 1986).

25. Rich, Elaine. Artificial Intelligence. New York: McGraw-Hill, 1983.

26. Seitz, Charles L. "The Cosmic Cube," Communications of the ACM: 28 22-33 (January 1985).

27. Siegel, Leah J. et al. "Performance Measures for Evaluating Algorithms for SIMD Machines," IEEE Transactions on Software Engineering, SE-8: 319-331 (July 1982).

28. Siegel, Howard Jay, and Robert J. McMillen. "The Multistage Cube: A Versatile Interconnection Network," Computer: 65-76 (December 1981).

29. Seward Walter D., and Nathaniel J. Davis IV. "Opportunities and Issues for Parallel Processing in SDI Battle Management/C3." Presented at the AIAA Computers in Aerospace V Conference, October 1985.

30. Stankovic, John A. et al. "A Review of Current Research and Critical Issues in Distributed System Software," Distributed Processing Technical Committee Newsletter, 7: 14-47 (March 1, 1985).

31. Stankovic, John A. "A Perspective on Distributed Computer Systems," IEEE Transactions on Computers, C-33: 1102-1115 (December 1984).

32. Stein, Kenneth J. "DARPA Stressing Development of Pilot's Associate System." Aviation Week and Space Technology: 69-74 (22 April 1985).

33. Treleaven, Philip C. "Parallel Models of Computation," Parallel Processing Systems, edited by David J. Evans. Cambridge, MA: Cambridge University Press, 1982.

34. Tuazon, J., et al. "Caltech/JPL Mark II Hypercube Concurrent Processor." IEEE Publication, 1985.

35. Wah, Benjamin W. et al. "Multiprocessing of Combinatorial Search Problems. Computer: 93-108 (June 1985).

36. Wu, Angela Y. "Embedding of Tree Networks into Hypercubes," Journal of Parallel and Distributed Computing, 2: 238-249 (1985).

37. Yourdon, Edward. Managing The Structured Techniques (Second Edition). Englewood Cliffs, New Jersey: Prentice-Hall, 1979.

VITA

Captain Richard T. Mraz was born on 16 November 1960 in Gloversville, New York. After graduating from Mayfield Central High School, Mayfield, New York in 1978 he attended the United States Air Force Academy. Upon graduation from USAFA in 1982 with an Bachelor of Science in computer science, he was assigned to the Air Force Data Systems Center (now 1st Information Systems Group), Pentagon, Washington, D.C. At the Pentagon, his primary duties included local area and long haul computer network analysis. In May of 1985, Captain Mraz entered the Computer Engineering program at the School of Engineering, Air Force Institute of Technology.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: Unclassified

1b. RESTRICTIVE MARKINGS:

2a. SECURITY CLASSIFICATION AUTHORITY:

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE:

3. DISTRIBUTION / AVAILABILITY OF REPORT: Approved for public release; distribution unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCE/ENG/86D-2

5. MONITORING ORGANIZATION REPORT NUMBER(S):

6a. NAME OF PERFORMING ORGANIZATION: School of Engineering

6b. OFFICE SYMBOL (If applicable): AFIT/ENG

7a. NAME OF MONITORING ORGANIZATION:

6c. ADDRESS (City, State, and ZIP Code):
Air Force Institute of Technology
Wright-Patterson AFB, Ohio 45433-6583

7b. ADDRESS (City, State, and ZIP Code):

8a. NAME OF FUNDING / SPONSORING ORGANIZATION:

8b. OFFICE SYMBOL (If applicable): OSD/SDIO

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER:

8c. ADDRESS (City, State, and ZIP Code):
Pentagon, Washington D.C.
20301-7100

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): Performance Evaluation of Parallel Branch and Bound with the Intel iPSC Hypercube Computer

12. PERSONAL AUTHOR(S): Richard T. Mraz, Captain, USAF

13a. TYPE OF REPORT: MS Thesis

13b. TIME COVERED: FROM ________ TO ______

14. DATE OF REPORT (Year, Month, Day): 1986 December

15. PAGE COUNT: 201

16. SUPPLEMENTARY NOTATION:

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number):
Parallel Processing; Parallel Search; Branch and Bound; Search; Object-Oriented Design; Hypercube

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Advisor: Walter D. Seward, LtCol, USAF
Assistant Professor of Electrical and Computer Engineering

Approved for public release: IAW AFR 190-1.
LYNN E. WOLAVER 7 April 87
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION: Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL: Walter D. Seward, LtCol, USAF

22b. TELEPHONE (Include Area Code): (513) 255-2024

22c. OFFICE SYMBOL: AFIT/ENG

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

Abstract

With the recent availability of commercial parallel computers, researchers are examining new classes of problems for benefits from parallel processing. This report presents results of an investigation of the set of problems classified as search intensive. The specific problems discussed in this report are the 'backtracking' search method of the N-queens problem and the Least-Cost Branch and Bound search of deadline job scheduling. The object-oriented design methodology was used to map the problem into a parallel solution. While the initial design was good for a prototype, the best performance resulted from fine tuning the algorithms for a specific computer. The experiments of the N-queens and deadline job scheduling included an analysis of the computation time to first solution, the computation time to all solutions, the speed up over a VAX 11/785, and the load balance of the problem when using an Intel Personal SuperComputer (iPSC). The iPSC is a loosely couple multiprocessor system based on a hypercube architecture. Results are presented which compare the performance of the iPSC and VAX 11/785 for these classes of problems.

END
5-87
DTIC