2

A Heterogeneous Parallel Programming Capability[1]

Final Report

J.W.Flower[2], A.Kolawa

ParaSoft Corporation
2500, E. Foothill Blvd.
Pasadena, CA 91107

Phone: (818)-792-9941
FAX: (818)-792-0819

Contract: N00014-90-C-0179
Topic Number SDIO-90-010
November 30, 1990

[1] Research sponsored by SDIO/IST and managed by ONR/NOSC under the SBIR program.
[2] Principal Investigator

Table of Contents

1. Statement of the Problem 1

1.1 Portability and Standardization 1

1.2 "Meta"-Computers. 2

1.3 System Integration. 2

1.4 Heterogeneous parallel processing with *Express*. 3

2. Phase I Objectives 4

3. Phase I Results 5

3.1 Implementation Strategy 5

3.2 Evaluation of network protocols 7

ISIS., 7

"Standard" RPC mechanisms., 7

Socket level communication, 8

3.3 Basic Interprocess Communication Design 8

3.4 Network Configuration. 9

3.5 Booting the Workstation Network 11

3.6 Implementation of *Express* runtime system. 12

3.7 A distributed debugger 13

Using `ptrace` on running programs, 13

Single stepping, 15

Alternatives, 15

3.8 Performance Monitoring 16

3.9 System Checkout 16

4. Conclusions and the Future. 17

4.1 User level "domains". 18

4.2 Object Oriented Engineering. 18

4.3 Support for external *Express* implementations. 19

4.4 Strongcr Language Support. 19

4.5 Integration of hardware and software "domains".. 19

Appendix A. Detailed "`domtool`" Documentation. 20

Statement "A" per telecon Dr. Keith
Bromley. Naval Ocean Systems Center/
code 7601. San Diego, CA 92152-5000.

1. Statement of the Problem

In creating a heterogeneous parallel processing capability we are really trying to approach three basic problems with current systems:

- Supercomputer and parallel computer hardware architectures vary widely but need to support one or two fairly standard programming languages and programming models. A particularly important issue concerns the short life cycle of individual hardware designs.
- Many algorithms require capabilities beyond the reach of single supercomputers but could be approached by several machines working together.
- Performing a given task requires integration of a "system" that may contain many components in addition to the super or parallel computer itself. Peripherals from many different manufacturers must be incorporated.

In addressing these issues we have developed several useful tools which have utility outside the range of the original plan. These will be discussed in connection with the "Implementation Strategy" in a later section.

1.1 Portability and Standardization

SIMD architectures such as Thinking Machine's Connection Machine have shown that enormous CPU power can be achieved from extremely simple components used in large quantities. Furthermore this architecture has been applied to a number of important applications with good results.

Distributed memory MIMD machines such as the hypercubes continue to achieve greater and greater speeds as technology increases the speed of the basic processor and the internode communication channels. These machines have built up a powerful body of successful results and continue to do so.

Shared Memory MIMD machines such as the Cray, Alliant, Convex, etc. are also becoming faster as the basic technology improves. Newer switch designs are achieving better results with lower memory and Bus contention. Programming tools have been developed that partly automate the process of generating multi-processor programs.

While these three categories of machine are, each independently, successful there is little standardization among the systems and as a result the most successful applications have often been developed for one particular class of machine, or even a single member of that class, through painstaking work by engineers and scientists. Once complete this application is then condemned to live out its useful life on a particular machine with little hope of being able to take advantage of the next series of hardware developments.

The last few years have, however, begun to see powerful efforts in the computer community to standardize many aspects of the software development cycle. Standards for languages such as Ada, ANSI-C and the new Fortran/90 are attempts to eliminate idiosyncracies among language dialects which hinder program portability. The emergence of UNIX has also gone a long way to giving programmers a standardized working environment. What neither cf these strategies provides, however, is a "standard" programming model.

Express is an attempt to fill this gap for parallel processing.

In assessing its success in fulfilling this role we note that *Express* currently runs on most of the important distributed memory MIMD machines and also some of the shared memory machines. It is in widespread use in a broad range of applications.

1.2 "Meta"-Computers

Although the advent of supercomputer systems has allowed many problems to be addressed that were previously believed impossible it has also promoted interest in solving new, even harder, problems. Computational chemistry and computer animation, for example, are fields which have arisen more or less as a direct consequence of the availability of supercomputer power.

Some applications, however, remain beyond the reach of even the fastest supercomputers. In particular defence related projects typically still require computer performance at least one order of magnitude higher than is available now. It is also to be expected that as more CPU power becomes available harder problems will be attempted.

One possible approach to this problem is the "meta"-computer concept as shown in Figure 1. The simple idea is that if parallel processing is accepted among homogeneous processing elements why not between supercomputers treated as nodes of a heterogeneous system? In this manner we can hope to address two important issues - providing performance greater than any single supercomputer and also better matching between algorithms and architectures.

1.3 System Integration

It is rare for the solution to any large problem to consist of CPU power alone. Typically I/O support, graphics and mass storage media are all involved in a complete solution. In advanced areas it may be required to interface complex custom designed hardware for data collection or analysis.- Components of the integrated system may also be required to control, in real time, pieces of hardware either locally or remotely.

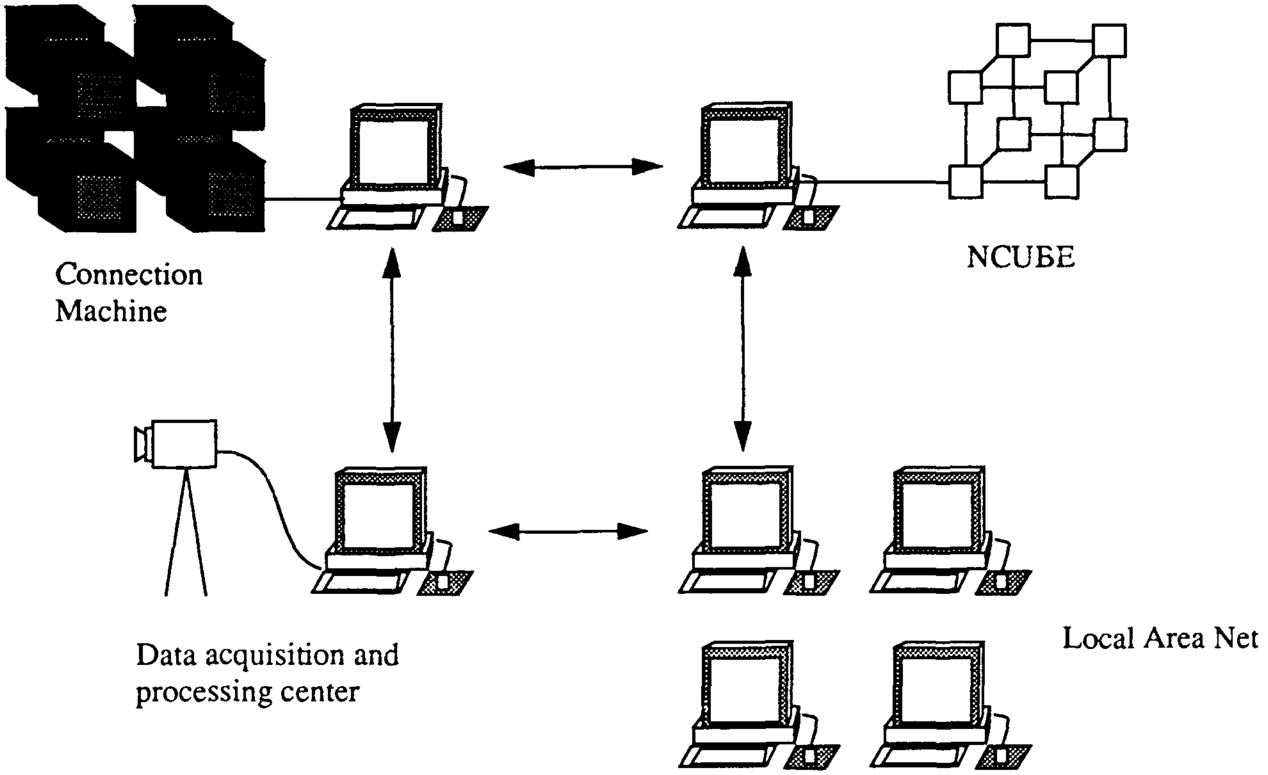


Figure 1 A "meta-computer"

It is unreasonable, however, to expect that supercomputer manufacturers will be able to support interfaces to all possible types of peripheral. Even though standardization has begun in some areas it has not yet reached the stage where any one protocol can be said to dominate. As a result we believe that supercomputer solutions can best be realized by using peripherals and CPU units as interconnected building blocks. In case one particular type of interface is unavailable it can be replaced by some combination of others. In this way a complex heterogeneous system can be built which offers solutions to the various CPU, I/O bandwidth, real-time, control and other constraints of the problem.

Once again, however, we are faced with the problem of controlling this system in a coherent manner. It is to be hoped that at the very least a portable programming model can be developed which allows the programmer some degree of uniformity across the various pieces.

1.4 Heterogeneous parallel processing with *Express*

Until now the various implementations of *Express* attempted to address only the first of these issues - providinr a portable, standard platform for parallel programming on a wide variety of different systems. Each implementation, however, was independent, but allowed programs to execute on a single parallel computer system while being source level compatible with other implementations.

The purpose of this proposal is to develop a "Heterogeneous parallel processing" capability in which programs could be written to execute simultaneously on several different parallel processing platforms potentially of different architectures physically located in widely separated sites. These programs would still be able to use the *Express* parallel processing paradigms and system calls and would be able to execute on much more limited resources when required but would offer the potential of extremely powerful "meta-computers" built up of communicating networks of supercomputers. We believe that such a system will become increasingly important, especially in areas such as SDI where real-time coordination of a vast number of processing tasks must be accomplished. The extra flexibility provided by the ability to combine different hardware architectures is likely to be of central importance. SIMD machines, for example, are extremely good at spatially homogeneous simple processing tasks such as early stages of image recognition. MIMD architectures, however, provide the flexibility to operate sophisticated AI systems. Combined with data collection equipment such a system might be expected to be the keystone of an SDI program.

2. Phase I Objectives

Our Phase I objectives were to investigate the requirements of a heterogeneous parallel processing system in an environment made up of standard workstations. By creating a working version of *Express* on this platform we would learn many of the features and problems to be expected when integrating other parallel computers.

Note that our approach in this regard is unique. Several other groups have tried to develop distributed programming environments. ISIS and Nectar are two such systems which concentrated on the workstation as the basic implementation platform. Neither system, however, has been implemented on a parallel processing platform and as a result their design goals and minimum requirements are rather different from ours. Because *Express* has already been implemented in the rather harsh support environment offered by supercomputer class parallel processing systems we have been forced to adopt programming models and minimum requirements which will more easily be able to support the eventual integration which is our goal.

The basic goals of our Phase I research were as follows:

- Evaluate possible networking models and existing distributed systems as potential candidates for the *Express* message transport system.

- Design a communication "server" which can support all of the requirements of *Express*.
- Provide a tool which allows users to design distributed heterogeneous systems.
- Implement the standard communication mechanisms in *Express* and the parallel I/O and graphics extensions.
- Implement the debugging and performance analysis systems to complete the standard *Express* system.
- Evaluate the system as a prototype for future development into a heterogeneous parallel system and, in particular, understand what modifications, if any, are required in the *Express* system.

We believe that we have satisfactorily completed all of these tasks. Many interesting lessons have been learned about both *Express* and the various features of standard workstations as they apply to parallel processing. We have developed a system which can execute on a network of simple workstations the same *Express* programs as run on various supercomputer parallel machines. To this extent the workstation environment can be viewed as a cheap and readily available platform for prototyping applications for execution on dedicated parallel hardware although it offers additional features which make it useful in its own right.

As well as answering the questions of our Phase I work plan we have found additional areas which require work and which were not appreciated originally. Some of these regard the efficiency of various pieces of the *Express* system while others concern the whole area of heterogeneous programming. We will discuss these in a later section and hope to pursue them in Phase II of this project.

3. Phase I Results

3.1 Implementation Strategy

The implementation of the heterogeneous version of *Express* builds around standard networking systems as its message transport medium. In this sense we are again fortunate that the standardization efforts of recent years have been so successful. The basic idea behind the system is that the various parallel machines making up the "meta-computer" will be linked together across an ethernet to which an interface is provided which supports all of the standard *Express* system calls.

To achieve this goal we make the assumption that each parallel processing system involved has at least one "point of attachment" to an ethernet. For the shared memory architectures this requirement is usually met trivially since each processor typically runs some version of UNIX. The distributed memory and SIMD architectures normally provide, at the very least, a "host" which can be connected to the ethernet and next generations will probably have special purpose peripherals designed solely for ethernet connectivity. We do not believe, therefore, that this assumption is restrictive.

Once each of the machines has a connection to an ethernet we can begin to set up a programming model. The simplest extension would be to allow, for example, nodes in a hypercube to be allocated to a program running on a machine other than the natural "host" for the system. This configuration would use the ethernet as an intermediary as shown in Figure 2. From this point it is simple

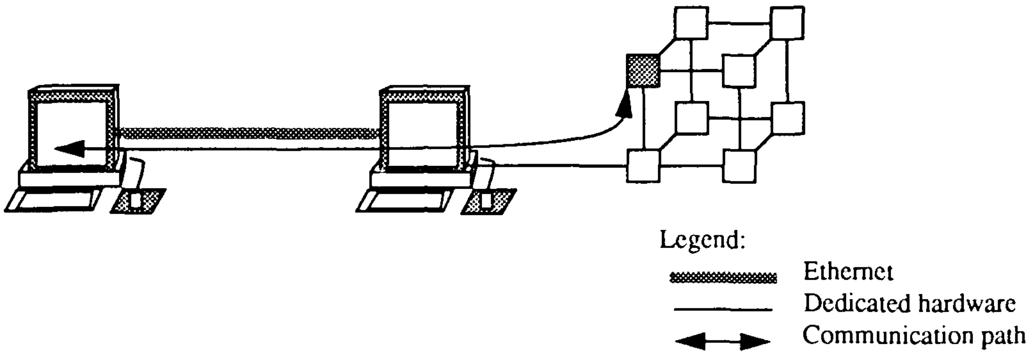


Figure 2 Remote access to hypercube nodes via network daemons

to make the extension which would allow two parallel computer systems to be connected and their nodes to communicate with each other, Figure 2.

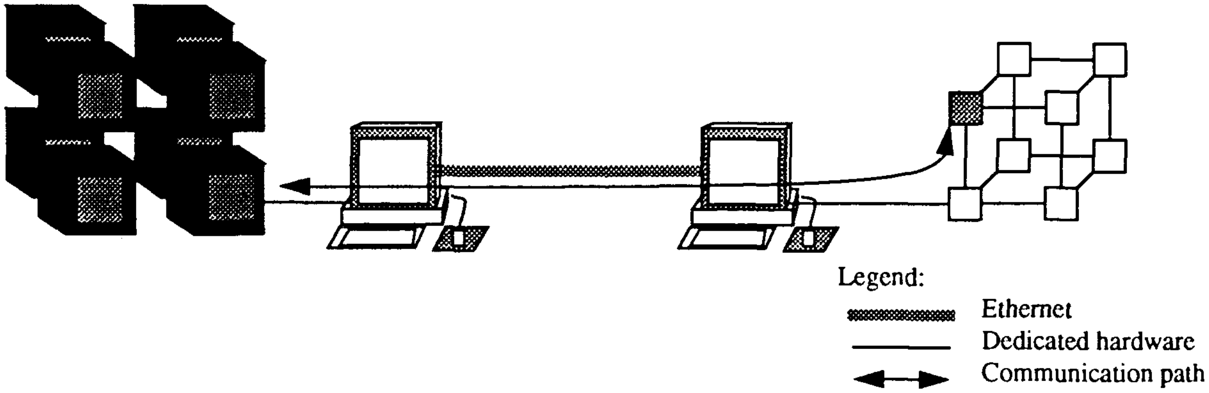


Figure 3 Interconnection of two parallel processing systems via servers

Phase 1 of this proposal requires the construction of a basic parallel processing capability among a group of ethernet connected workstations. The technology required to build such a system is clearly the major part of that required to realize Figure 2.

The workstation system is also useful in its own right. The explosive growth in the workstation industry has led to a situation in which there are often as many people as machines in many organizations. Furthermore the CPUs of many of these machines are often idle or very underloaded. Considering that the CPU power of many of the advanced workstations rivals that of many of the older parallel computers there is an enormous amount of computing power going to waste.

One mechanism for utilizing this "waste" is to use "distributed" programming methods such as those embodied in Linda and Strand88. These programming methods are able to take advantage of wasted CPU cycles by operating in "master-slave" modes in which work is assigned to idle processors on a network. These methods are suitable for some types of problem but suffer from weak implementations on dedicated parallel processing hardware. As a result the programs generated do not scale well when "real" hardware becomes available.

An alternative option is to use a genuine "parallel processing" model such as *Express* in the workstation environment. This approach shares with Linda and Strand88 the advantages of simplicity and portability but also leads naturally to extremely efficient implementations on full scale hardware.

3.2 Evaluation of network protocols

We evaluated several different types of network software when deciding on the lowest levels of the *Express* system.

3.2.1 ISIS.

Carnegie Mellon's ISIS system is a message passing environment for use on large workstation networks. As such it corresponds to the lowest level functionality of the *Express* system. A particularly strong feature of the system is its availability for a wide range of workstation types and the simple "fault tolerance" concepts which have been developed.

Unfortunately the current implementation is not compatible with the multi-threaded approaches required to implement some of the *Express* features and we are unable to take advantage of the system. We intend to periodically review this decision.

3.2.2 "Standard" RPC mechanisms.

We also examined the two leading RPC mechanisms (Sun Microsystems and the Open Software Foundation) as a basis for our system. Both offer similar functionality for our purposes and could be used to generate tasks on remote workstations. The biggest problem with using this type of software for our purposes is the overhead involved in the RPC parameter passing mechanisms which are much too general for our needs. Furthermore the programming model typically assumes that the spawning program will not need to communicate with the spawned process other than to pass it its initial arguments and to collect its results. As a result the process of establishing a communication channel to the new process is no simpler than would be the case for a much lower level ethernet interface.

3.2.3 Socket level communication

The simplest way of maintaining and using an ethernet connection is to use the "`socket`" system calls and either TCP/IP or UDP datagrams.These methods impose no restrictions on the types of data to be transmitted and have much lower overheads than the RPC mechanisms. They are, however, just as portable. The choice between the two network layers, TCP/IP and UDP, is again basically one of overhead. The advantage from our point of view of the TCP/IP layer is that it guarantees delivery of network packets in the order in which they are sent, providing, of course, that the target machine does not fail. On the other hand the UDP transport mechanism is much faster.

We have currently adopted the TCP/IP layer for our message transport but have isolated the appropriate code in modules with well-defined interfaces so that we can, at a later date, adopt either the UDP ethernet protocol or the ISIS methods.

3.3 Basic Interprocess Communication Design

In practice it was discovered that the requirements of the basic interprocess communication played an important role in the decision of basic network transmission protocol. The difficulties inherent in the two processes discussed below, for example, we responsible for our decision not to use ISIS as the basis for our communication strategy.

One of the strong features of *Express* is its support for "message interrupts". The `exhandle` system call allows a standard message to trigger an immediate response from a function embedded in the program executing on the target node. This feature allows us to support "real-time" operations modes as well as a remote procedure call mechanism and multitasking *all based on messages as the fundamental operation.*

Support for this features requires that the program running at the target node be interrupted when an appropriate message arrives and the necessary "handler" routine invoked to process it. To this extent a "multi-threaded" programming model is required. (Note that a multi-threaded *operating system* is not required, only the ability to interrupt a user application.) Most operating systems support this type of functionality either by means of the UNIX "`signal`" mechanism or some form of "lightweight process" library.

Of these alternatives the latter is more attractive since it uses (presumably) optimized context switching. Unfortunately, however, most current implementations offer only non-preemptive scheduling and/or suffer from the problem that if any thread in a process sleeps the entire user process sleeps. Neither is suitable for *Express*. The "`signal`" method, on the other hand, suffers from the known defects of the UNIX signal procedures. Nevertheless it is the only solution which fills the needs of *Express* and has been adopted for our use.

Since the area of multi-threaded applications is one which is receiving a lot of attention currently we will continue to review progress in the field for better solutions which match our requirements.

The second design feature which motivated our choice was the desire to use optimized "shared memory" transfers for communication between processes resident on the same workstation. Obviously using this medium for communication is much faster than going through the ethernet protocols.

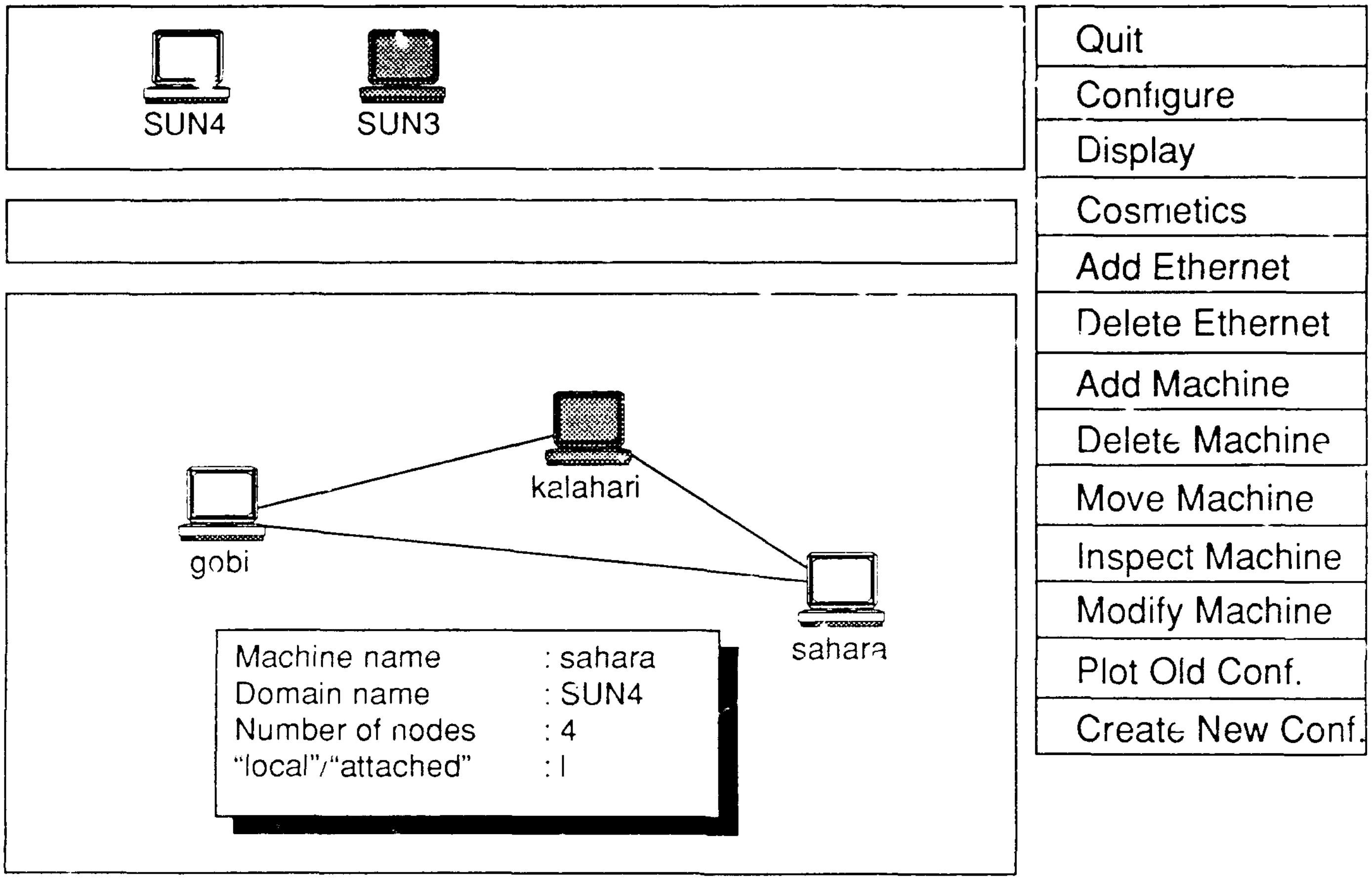
3.4 Network Configuration

The question of network configuration is central to the heterogeneous computing system. Unless the system consists of processors of more or less equal capability we are faced with the task of assigning work to processing units with widely varying strengths and weaknesses. This presents problems in load balancing and other areas but also opens up many interesting possibilities for the user. As indicated in the opening comments to this report one of the interesting and important goals of our research is to provide parallel programs with facilities not easily achieved by using a single type of parallel processor. In this case we must allow the user to take advantage of facilities provided by the differing capabilities of workstations on the network.

One common use of such a capability is in high performance graphics applications. Many workstations possess sophisticated systems for viewing objects but may not possess the "number crunching" power to generate the necessary data. Other useful capabilities such as video interfaces or CD-ROM devices may only be available on a restricted set of workstations.

To deal with these cases we must build into the heterogeneous system the ability to take advantage of such capabilities in a user friendly manner. Also, for administrative reasons, we need a facility for overall control of the workstation resources used by *Express* programs, in particular the network bandwidth and connections.

To approach these two problems we built a graphical utility, "`domtool`", which allows workstations to be added and/or removed from the network and also allows the specification of ethernet connections between machines. A sample of the display is shown in Figure 4. The basic idea is that



Machine name : sahara
Domain name : SUN4
Number of nodes : 4
"local"/"attached" : l

Figure 4 Use of the "`domtool`" configuration utility

as each workstation is added to the display a series of questions are asked regarding its use. These questions are basically:

- The ethernet name of the workstation. This is required only when booting the system.
- The "domain" to which the machine will belong. This is a concept relating directly to the "heterogeneous" nature of the workstation environment and is discussed below.
- The maximum number of "nodes" to be attached to this workstation. Note that these are "nodes" the sense previously described.
- Whether these nodes are "local" or "attached". This difference concerns the nature of the "nodes". If they belong to a genuine parallel processor they should be described as "attached" If they are to be executed on the workstation itself they

should be described as "local". This difference may disappear as the system evolves.

The "domain" concept was developed in response to our debugging needs and exceeds the bounds of the Phase 1 proposal. Originally the first phase work was intended to be solely on the workstation/ethernet aspects of the system with the bulk of the "heterogeneous" topics delayed until Phase 2. In debugging the networking interface, however, we soon had to deal with a network composed of both Sun-3 and Sun-4 workstations - a heterogeneous system.

Our solution to these problems is to create a secondary classification scheme for machines; domains. The only real restriction on a domain is that all the processors in a single domain must be able to execute a given executable file if any of them can. A domain can, for example, contain several Sun-3 machines or several Sun-4s but not a mixture of both.

A more subtle use of the system is to categorize machines by their capabilities. A domain might be constructed consisting of all the nodes with local disks or special purpose graphics hardware. In this case the set of Sun workstations of a given CPU type might be further subdivided.

The use of this concept allows the application programmer several levels of sophistication in running parallel programs. At the most detailed level individual workstations can be specified by name for the nodes of a parallel program. At the opposite extreme no specification can be made in which case *Express* attempts to find any workstation in the system to execute a given program. The intermediate (and most useful) stage allows the nodes of the user program to be mapped to the "domains" defined by "`domtool`". This maintains the flexibility and reconfigurability of the latter approach while still allowing *Express* to make decisions about process assignment.

As well as assigning domains "`domtool`" serves another important purpose - the allocation of network bandwidth to interprocessor communication channels and "socket" resources.

Since ethernet bandwidth is a fairly important commodity *Express* allows restrictions to be placed upon the particular channels it uses. No assumption is made about the connectivity of the underlying network. Further "socket" connections between workstations are only made as indicated to "`domtool`". Workstations which are required to communicate but which have no direct socket connection use the forwarding system built into *Express*. This restriction is required, especially for large systems where it is impossible to open enough sockets to fully connect a large network.

3.5 Booting the Workstation Network

Once the network has been "designed" with "`domtool`" it is booted with the "`exinit`" command familiar to users of *Express* in other contexts. This command is responsible for setting up the desired socket connections and downloading information to "server" processes started on each workstation. Among the data required to operate are

- The "forwarding" table which describes the paths through the sockets between any pair of workstations.
- The "domain" map which assigns workstations to domains used for loading user applications.
- The processor numbers and other *Express* related information.

The booting process is the one location where we use the RPC commands, albeit indirectly. The UNIX "rsh" command is used to spawn the appropriate daemons.

Again this decision represents a compromise. In practice setting up a network to use "rsh" in the correct mode, with the correct privileges is a slightly non-trivial process which we have found in field tests to be quite tricky. A potentially better alternative is to use the RPC mechanisms directly by locking special ethernet "port addresses" to the *Express* daemons. This solution also has its problems in the setup phase but might in the long run offer more attractive features than the "rsh" mechanism in use now.

One of the most challenging problems in booting the system concerns the issue of abrupt program termination. In a smoothly functioning environment it is easy to construct servers which correctly respond to user programs. Unfortunately, however, we are faced with a development environment in which some users will be running "functioning" parallel programs while others are developing new algorithms with their associated "bugs". As a result the system of servers must be able to cope with programs that "disappear" as they are aborted by their users or which "crash" due to other problems. This is a rather weak area for UNIX but we are able, by combining various "signal" mechanisms with periodic "cleaning" processes to deal with most cases.

3.6 Implementation of *Express* runtime system

Once the underlying message transport mechanisms had been completed we proceeded to implement the standard *Express* communication utilities which make up the majority of the user-visible *Express* programming model. In addition to the basic communication utilities we also implemented the *Cubix* I/O system and the *Plotix* graphical system.

The implementation of the runtime system proved to be relatively straightforward once the underlying protocols had been developed. In particular the implementation of the exhandle, exsend and exreceive system calls was easily achieved by using the UNIX signal mechanisms. This had been included in the original decisions that led to our choice of networking protocols and server designs and so caused few additional problems.

At the completion of this work we were able to execute the standard set of *Express* test, example and demonstration programs on the workstation network.

3.7 A distributed debugger

The implementation of the distributed debugger, `ndb`, proved to be rather more troublesome. In general the implementation of the debugger is a two stage problem:

- Implementation of code which parses the compiler generated symbol tables and builds appropriate data structures.
- Support at the kernel level for single stepping, breakpoints, etc.

In our experience with dedicated parallel processing systems the former is often quite tricky because the compilers are often non-standard or are early prototypes that follow no conventions as regards symbol table formats. Often it is difficult to extract the information required by `ndb` from the executables, even if it is present. In contrast supporting memory reads/writes, breakpoints and other standard debugging features is often simplified by the extremely simplistic nature of the operating systems on such hardware. In most cases it has proved simple to add the necessary support to existing kernels.

In the workstation environment we originally hoped that the process would be doubly simple. UNIX typically supports fairly standard symbol table formats and has well matured compilers. Furthermore the availability of high-quality debuggers for sequential programs led us to hope that we would have a relatively easy task in implementing our debugger.

The situation with the symbol tables was indeed fairly straightforward. Documentation was easily forthcoming and the formats are well understood and reasonably well thought-out. (The one exception being Sun's `dbx` format which is extremely verbose and clumsy.)

Our current problems center on the use of the `ptrace` system call as the entry point to UNIX's debugging mechanism. In principle this supports all of the features that we would expect but it suffers from several defects as explained below.

3.7.1 Using `ptrace` on running programs

As implemented on most workstations the `ptrace` system call is only supported when the program being debugged is "stopped". In a conventional debugger this causes no real problems since the debugging cycle is typically "insert breakpoints, examine variables, continue execution". Because of this cycle the user only examines the state of the target application when it is stopped at a breakpoint or when it has “dumped core” and `ptrace` is sufficient.

In a parallel program, however, much more complex situations arise. Consider, for example, a rather contrived case as shown in the following “pseudo-code”.

```
Node 0                                 Node 1

     10   foo = 100;                        10   foo = 200;
     11   bar = 100;                        11   while(1);
     12   joe = 100;                        12   joe = 300;
```

In this example we assume that due to some extremely odd “bug” the program in Node 1 will never reach line 12. Let us assume that the programs in both nodes are currently stopped at breakpoints at line 10. The user inserts a breakpoint and line 12, examines various variables and finally asks the program to continue on both processors.

At this point the program in node 0 will execute line 11 and then stop at line 12. The program in node 1, however, will execute forever at line 11, never reaching its breakpoint.

In standard sequential debuggers this situation would result in the user never being given back a command prompt. This is reasonably acceptable in a sequential debugger since the user is aware of the problem and can interrupt the program with some keyboard command to see what is happening.

In this particular case this solution may be vaguely acceptable in the distributed debugger since, after all, node 1 is never going to reach the breakpoint and the user program has no chance of finishing. On the other hand a much better solution would be to tell the user that node 0 has reached the breakpoint and that node 1 is still executing code. The user would then be in a much better position to try and find the problem. In more complex cases where several nodes are each waiting for each other in some complex sequence it is vital that the user be able to examine the processing state even when some of the nodes are still executing code.

Unfortunately the standard implementation of the `ptrace` system call does not support this capability. When requests are made of a currently executing process the results returned by `ptrace` are typically wrong, often with no indication of an error.

At this point our solution to this problem is to implement a “weak kill” mechanism by which one of the server processes sends a UNIX `signal` to a process whose state is requested by `ndb`. This causes the process to halt and thus generate correct responses to the `ptrace` call once account has been taken of the fact that the user program was interrupted by the debugger itself asking for information. Once the response has been received the server restarts the node process.

This process is, unfortunately, both slow and error prone. The use of UNIX `signals` is complex and non-intuitive with many different combinations of events causing unpredictable behavior. At this time we are attempting to find alternative solutions to this problem.

3.7.2 Single stepping

A second area of difficulty in connection with the distributed debugger concerns the single-stepping process. Even in debuggers for sequential programs this can be quite a slow process depending on the degree of support from the underlying hardware and the complexity of the user program. In the parallel case, however, other problems occur.

Consider the following program fragment:

```
Node 0                                Node 1

    10   foo = 100;                       10   foo = 200;
    11   sync with node 1;                11   sync with node 0;
    12   joe = 100;                       12   joe = 300;
```

Again we assume that the program is stopped at a breakpoint in both nodes at line 11. If we now try to single step both nodes to line 12 a potential problem arises.

The sequential debugger carries out a single-step by continually executing machine instructions until reaching the next line number in the user program. If we apply this procedure to the code in node 0, however, we will reach an impasse because the program in node 0 will never reach line 12 - it will wait to synchronize with node 1 which is still stopped at its breakpoint.

A solution to this problem is to alternate the low level single stepping process between the affected nodes. In this way we will eventually meet the synchronization criterion and reach line 12.

The penalty for this approach, however, is its speed. Communicating with the *Express* servers on each node is a high bandwidth, high latency process because of the nature of the underlying Ethernet. Since the control messages used by the debugger are short we have only to contend with the high latency and as a result the single stepping process can become prohibitively slow.

At the present time we are exploring possible alternatives to this approach.

3.7.3 Alternatives

At the present we are somewhat disappointed by the implementation of the `ptrace` system call. While offering all the capabilities required by a debugger for sequential programs it is seriously lacking in support for more complex applications. An additional problem is that different machines tend to support this facility in subtly different ways which make this aspect of our system less portable than we would like. Almost the only aspect of the ptrace interface that is common is its refusal to allow debugging of running processes!

An alternative implementation of the debugging process is implemented on a few UNIX like systems. A "pseudo-device" is created for each running process which can then be addressed in a manner reminiscent of a standard file. In particular one can read and write the memory of a process using the conventional UNIX read and write system calls. We are particularly interested in this approach since it offers potential to solve at least the first of the problems discussed in this section. In future it may also be possible to implement local interprocess communication through this mechanism which will provide an alternative to the shared-memory implementation adopted in Section 3.2.

A task for further research is to evaluate this mechanism in regard to both debugging and use as a standard messaging protocol for *Express.*

3.8 Performance Monitoring

The final part of the standard *Express* system to be implemented on the workstation network is the performance analysis system, *PM.* Again the implementation of this process proved relatively straightforward. The "communication" and "event-driven" systems required only simple "porting" similar in nature to that required to implement the basic high-level communication system. The "execution" profiler relies on periodic interrupts in the same manner as the conventional UNIX profilers "prof" and "gprof" and was implemented in the same manner.

3.9 System Checkout

We have evaluated the system, at least for correctness, by executing the standard set of test, example and demonstration programs as are used for *Express* check-outs on dedicated parallel computer systems.

We have found that the programs execute without modifications to the source code - a powerful exhibition of the source code compatibility of the *Express* system.

In terms of performance we have, as yet, little idea of the absolute capabilities of the workstation environment. Already apparent, however, is the fact that we can usefully use a network of Sun-4 class machines to develop parallel applications in the absence of dedicated hardware. In some respects this is actually preferable to the real target system - the compilers are more mature and execute much faster than the cross-compilers typically used for genuine parallel hardware. This shortens the compile/run/debug cycle and also relieves the developer from struggling with the compiler "bugs" often found in immature parallel processing software. The graphical capabilities of the workstation environment also make it an attractive alternative to parallel processing hardware which may support only very primitive operations.

We have also found that a single workstation can support up to four "node" programs without becoming too overloaded to perform useful work without impacting its neighbors on the local area net.

The issue of the absolute performance of these systems will probably remain an open question until an interface to a real parallel processing system is constructed and a "heterogeneous" system is built. Only in this context will we be able to examine properly the trade-offs between the processing speeds and bandwidth requirements of the genuine parallel processing systems and the interface "glue" provided by the workstation environment. Also important in this evaluation are "load balancing" issues which are addressed more closely in another SBIR project currently being undertaken by *ParaSoft*. For this and other reasons we believe that the Phase II work carried out on this project should be combined with that of our other current SBIR award since important aspects of the work on the two projects overlaps.

4. Conclusions and the Future

The simplest conclusion realized by the Phase I research is the delivery of an implementation of the *Express* system that can run on a network of standard workstations. We have completed this project and have been able to implement all of the features of the other versions of *Express* which run on dedicated parallel processing hardware. In performing this work we have learned some of the strengths and weaknesses of the workstation environment and the UNIX support tools. In all cases we have found appropriate solutions to meet our needs although some of these are still undergoing study to see if improvements can be made.

The degree to which this project has been successful is indicated by the fact that we are able to run, *without changes to the source code*, all of the standard *Express* test and example programs. Even further we were able to compile and execute a significant adaptive grid finite element fluid mechanics solver without source code modifications.

The workstation system has been shipped to several *Express* user's and we are currently awaiting feedback on the strengths and weaknesses of the design.

The idea of a "domain" was a development that arose early in our research and had fundamental repercussions throughout the project. Originally we had intended that this concept be left undeveloped until reaching Phase II and real "heterogeneous" systems. In practice, however, we discovered that even a network composed of Sun-3 and Sun-4 class machines is genuinely "inhomogeneous" and requires more advanced support. The creation of "`domtool`" was a major effort in making the workstation system useful.

The most exciting results of this Phase I research, however, are the new and unexpected areas which have emerged. When originating this project we expected to create versions of *Express* that, while heterogeneous, shared the same programming tools and models of the original parallel processing systems. In fact what has happened is that in examining the requirements of the heterogeneous systems and its potential users we have found exciting new ways to extend the older versions of *Express* and improve its internal structure. We intend to explore these issues more fully in our Phase II proposal but some of the basic items are discussed below.

4.1 User level "domains".

The concept introduced at the current level is basically that of a hardware distinction between the capabilities of various machines. We believe that this must be supported by a corresponding software concept which will allow the user to partition their original algorithm into groups of processing elements which share a common task. These "groups" may or may not correspond to the hardware "domains".

This concept represents a middle ground between pure "domain decompositions" in which the same program executes on each node and the data is distributed and pure "functional decompositions" in which a different program or task executes in each node. The concept seems to be appropriate to the structure of the heterogeneous networks and should be supported at all levels of the *Express* system from the communication system, through the I/O system, to the automated parallelization system.

4.2 Object Oriented Engineering.

Current versions of *Express* are written with what may be called "old-fashioned" style code. Extensive use is made of external variables and common routine names to simplify the porting of the system between architectures. While this has proved valuable in the individual *Express* implementations it will cause problems in the heterogeneous environment where a single user program may require to be interfaced with several different runtime *Express* systems according to the types of hardware in the heterogeneous system under its control.

We intend to adopt an "object oriented" design philosophy in which types of hardware are abstracted and a clear distinction is made between code common to all implementations and machine specific details.

4.3 Support for external *Express* implementations.

All of the current versions of *Express* were generated either by company employees or by working in close collaboration with other organizations. This has proved effective to date since the number of dedicated parallel processing systems is small. Now that the system is being extended to cover not only workstation interfaces but also arbitrary combinations of machines there are far too many variants to carry out all the necessary work internally.

It is our intention, therefore, that *Express* become an "open system" for which the source code will be made available to users with novel hardware architectures. To facilitate this process we have to create a much "cleaner" development system than is currently used internally. We intend to build a semi-automatic "porting" methodology that should ensure that the system can be integrated with new hardware at minimal expense. This will require extensive automation of the internal procedures used to create the tools and runtime support and major documentation.

In conjunction with this idea we intend to setup an advisory committee drawn from the *Express* users who will have responsibility for developing the extensions and improvements to the system that will inevitably be generated by other groups working with the *Express* source code.

4.4 Stronger Language Support.

The *Express* system is itself written in C although the tools and runtime support are available to Fortran programmers. The techniques by which the Fortran interfaces are derived is currently somewhat time consuming and error prone and requires extensive improvement. This is particularly so since we intend to support other languages in the future such as ADA.

4.5 Integration of hardware and software "domains".

The "domain" concept must be extended to cover all of the software tools in use with *Express*. In particular programs such as the various profiling and debugging tools must be taught about the underlying hardware structures. It is our intention to attempt to conceal the underlying hardware configuration of the heterogeneous systems just as we currently hide the hardware topology of the various parallel processing systems.

This is of particular importance when debugging - we aim to provide a system in which the physical CPU type on which a process is executing is invisible to the user. This involves developing techniques by which the hardware specific features of all the software tools can be mixed interchangeably at runtime. As well as requiring a more object-oriented approach as described in 4.2, we must also integrate some sort of data-base of hardware types and capabilities.

Appendix A. Detailed "domtool" Documentation

This appendix contains the full documentation for the "domtool" domain configuration utility.

1 Introduction

To this point the networks used with *Express* have been "homogeneous" in the sense that a program, once compiled for the machine, can be executed on any of the attached nodes. Examples of this type of system are hypercubes and transputer arrays. With the exception of the host computer the nodes are all of the same basic architecture and can usually be treated as a homogeneous system. However, consider the network show in Figure 1. This

Heterogeneous networks

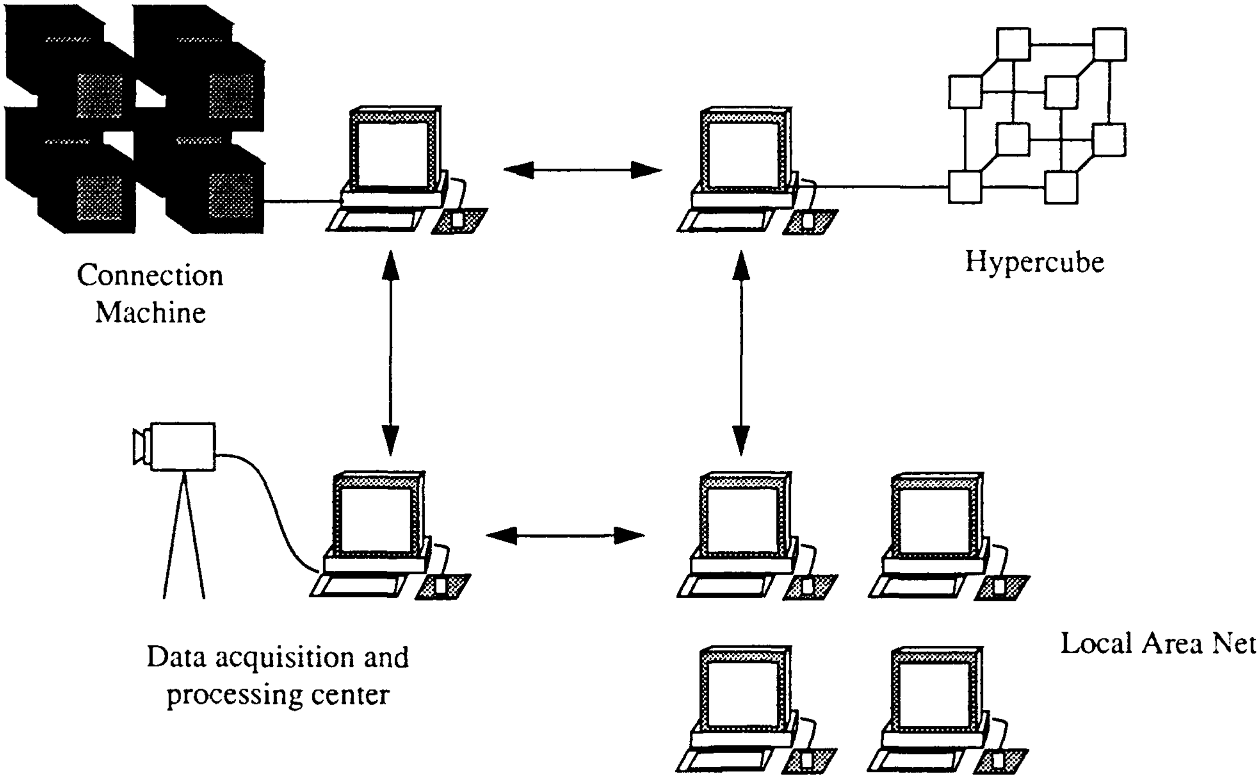


Figure 1 An Inhomogeneous Parallel Computing System

system can be used for parallel processing but can, by no means, be called homogeneous. The "node" programs that execute on the Connection Machine are completely different from those that execute on the hypercube. Similarly the programs that run on the workstation network will be different, not only from the executables for the other types of system, but may also vary from one workstation type to another. As well as these fundamental, but straightforward, differences between "node" types we are also faced with a much more complex programming problem as the capabilities of the components of the parallel computer differ widely from one node to the next.

2 Domains, Processors and Nodes

To deal with such complex issues *Express* uses the concept of "domains". A domain is a group of parallel processing nodes with the same hardware architecture. In Figure 1, for example, the hypercube system would be a single domain containing multiple nodes while the Connection Machine would be a domain with only a single node. The workstation network might be a single domain if all the machines were of the same type, or it could be several domains - one containing Sun-4 machines, one with Sun-386i's, etc.

Definitions: terminology becomes complex for heterogeneous systems

Before proceeding to a discussion of the programming model used on such systems we should first clarify the terminology to be used. This is made complex by the fact homogeneous systems are normally described in *Express* as having a "host" and one or more "nodes". While this concept is suitable for simple systems it requires extension in more complex situations. As a result let us define the following terms:

Domains

Domain — A domain is a group of machines of the same hardware architecture. This is a logical concept introduced to facilitate programming heterogeneous systems and is assigned when configuring an *Express* system. The most important factor in assigning domains is that an *Express* executable which can run on one machine in a given domain should be able to execute on all machines in that domain.

Processors

Processor — The individual unit of computation, at the level of the underlying hardware. A hypercube, for example, may have 1, 2, 4,..., 512,... "processors" each capable of running programs. A workstation, by virtue of its multi-tasking operating system may be configured to have any number of processors since it can run several programs at once.

Program

Program — In conventional systems a "program" is a particular piece of code that performs a given task. In parallel and distributed systems we use the term "application" for the overall structure and "program" for the individual pieces. On a parallel computer, for example, we use the term "program" for the code that runs in each processor. In a distributed system a "program" is the unit which is distributed between the processors.

Node

Node — This term should be taken to mean a single "processor" assigned to a user's parallel processing "program". This is a logical concept - not all the "nodes" assigned to a user need be of the same type or come from the same "domain". When running on 16 "nodes", for example, we may chose to use 8 processors from a hypercube and 8 from a transputer array.

Machines

Machine — A "Machine" is a named entity that can be assigned to a domain and which carries with it a specific number of processors. The name of the machine will be used in configuring the machine and is usually related to the "network" properties of the system. A hypercube hosted by a workstation with name "hyper-host" would be considered a single "machine" with this name.

While these definitions may seem needlessly pedantic (especially when considering the vague terms in which they are expressed) they may help to resolve confusion which may arise when configuring and using the systems.

In general terms we may say that

- A "domain" consists of one or more "machines".
- A "machine" may have one or more "processors".
- The user may allocate and work on any number of "nodes" distributed in any way among the "domains".
- The user's application is built from a number of "programs" which can be distributed among the "nodes".

Domains and "workstations" are normally synonymous

In most cases the term "machine" in the above descriptions is equivalent to "workstation". Most types of parallel computer are supported by a single workstation, the Connection Machine's "host", the "host" of the hypercubes or transputer arrays, the individual components of an *Express* workstation net etc. These machines are usually connected to an ethernet for communication purposes and their network known name is the identification parameter used in assigning machines to domains. The confusing issue is the number of "processors" connected to a particular workstation and the number of "programs" it can support. To clarify this issue let us define two more terms:

"Attached" processors have their own CPUs

Attached Processors

These are computing elements other than the workstation's CPU itself which are used for parallel processing. The individual processors of an associated hypercube array, for example, are "attached". This number is fixed by the hardware of the system.

"local" processors use the CPU of their workstation

Local processors

This is a logical concept which describes the number of *Express* programs which can simultaneously execute on the CPU of the workstation. This number can be chosen while configuring *Express* and is independent of hardware considerations. (Other than the fact that allowing hundreds of processors will slow down the machine quite a lot!)

An important distinction that *Express* makes in regard to this concept is that a single "machine" or workstation may not support both "attached" *and* "local" processors. This does not mean that a program cannot simultaneously use a hypercube and a workstation net under *Express* but the particular workstation supporting the hypercube cannot execute *Express* programs on its own CPU while it is playing the role of "host" for the attached nodes.

Having defined "local" and "attached" processors in this way we can naturally extend the concept to the "programs" which will be run there. A machine with 10 "local nodes" is a workstation which will support up to 10 simultaneous *Express* programs on its own CPU. Conversely a machine with 512 "attached node" can support up to 512 simultaneous *Express* programs on its attached processors but none on its own CPU.

3 Running Programs on Heterogeneous Systems; DDF Files

Basic program control on the heterogeneous net is achieved by modifying the allocation and loading strategies usually used by *Express* programs. A conventional *Express* "host" exopen *and* exload *do not support the full capabilities of the heterogeneous network.*

program might use a sequence of instructions similar to

```
if(pgind=exopen("/dev/transputer",4,DONTCARE)<0){
    fprintf(stderr,"Failed to allocate nodes\n");
    exit(1);
}
if(exload(pgind, "myprog") < 0) {
    fprintf(stderr,"Program load failed\n");
    exit(2);
}
```

to load the same program into every node. Similarly a *Cubix* program might be loaded with the command

```
cubix -n16 mytest
```

On the heterogeneous network life is not normally so simple. The above commands will still work and should do the expected thing - in the first case a program called "myprog" will be loaded onto four identical processors while in the second "mytest" will be loaded into 16 nodes. In each case the allocated nodes will be taken from *a single domain*, and furthermore, *Express* will try to figure out a suitable domain from the type of executable being loaded.

exnopen *and* exnload

While this is a reasonable default the heterogeneous net offers many opportunities which are not available in homogeneous systems. To take advantage of these alternatives *Express* offers a slightly different set of allocation/loading primitives which use a "domain description file" (DDF) to map "nodes" onto "processors"

Domain description files.

The basic idea of the DDF file is to tell *Express* what type of processors you would like to use in your parallel program, and the name of the appropriate executable for *Express* to load. Consider the following "DDF", for example.

```
# Sample domain description file.
0-3 SUN4 mynode4 Testing domains
4-6 SUN3 mynode3 on different hardware
7 GRAPHICS mygraph 512 512
```

Figure 2 Sample "Domain Description File"

DDF entries

The first line, starting with the '#' character is a comment, ignored by *Express*.

The second line specifies that nodes 0, 1, 2 and 3 should be allocated from the domain named "SUN4" and loaded with the program "mynode4". In addition, arguments "mynode4", "Testing" and "domains" will be passed to the node program at runtime.

The third line indicates that the program "mynode3" should be loaded into nodes 4, 5 and 6 and passed arguments "mynode3", "on", "different" and "hardware". These nodes will be allocated from domain "SUN3".

Finally the last line tells *Express* to allocate node 7 from domain "GRAPHICS" and load a third executable passing it the indicated arguments.

In this example we have used domain names that strongly suggest underlying hardware differences. We might imagine, for example, that domains "SUN4" and "SUN3" were made up of workstations of the indicated types while the "GRAPHICS" domain were built from machine(s) with special capabilities. This is not necessary - domain names may be completely arbitrary: "DOG", "CAT" and "HORSE" would be just as acceptable if somewhat less informative.

The DDF may be arbitrarily complex. Comments, continuation lines, quotes, etc. are all processed appropriately. Furthermore the nodes may be listed in any order and domains may be specified more than once. The following DDF loads all but one node with the program "complex", passing no arguments and allocating from domain "SUN4".

```
# Sample domain description file.
0-11 SUN4 complex
12 SUN3 mynode3 on different \
            hardware "this is one argument"
13-15 SUN4 complex
```

Figure 3 Sample "Domain Description File"

Node 12 is allocated from domain "SUN3", loaded with the program "mynode3" and passed 5 arguments of which the last is the string "this is one argument".

Using exnopen *and* exnload *in a "Host-Node" program*

To use this technique in a "host-node" program we use the routines "exnopen" and "exnload" in place of the more common "exopen" and "exload". If we have stored the domain description of Figure 3 in the file "domain.ddf", for example, we might use the code

```
/*
 * Allocate and load nodes in a host program, using a
 * domain description file.
 */
#include <stdio.h>
#include "express.h"

main()
{
     struct nodenv nodedata;

     if(exnopen("domain.ddf") < 0) {
         fprintf(stderr, "Failed to allocate nodes\n");
         exit(1);
     }
     exparam(&nodedata);
```

```
            if(exnload("domain.ddf") < 0) {
                fprintf(stderr, "Failed to load programs\n");
                exit(2);
            }
                    ...
```

The exnopen and exnload system calls each have a single argument, the name of the domain description file which describes the nodes to be allocated and the programs to be loaded.

Finding the number of nodes allocated

Note the inclusion of the call to exparam in the previous code. Normal host programs often do not need this since the number of nodes to allocate/load is a parameter to the exopen call and is usually stored in some program variable anyway. In the DDF case, however, the number of nodes to be allocated is determined directly from the domain description file and is not specified by a program variable. A good solution is the call to exparam (*after* the call to exnopen) which will return the number of allocated nodes in the nprocs element of the nodenv structure. This method is preferable to "hard-coding" the number of processors in the program since it will be determined at runtime from the DDF. Changing the number of processors in use then requires changes only in the description file.

Using DDF files in Cubix programs

Programs that use the *Cubix* programming model can be loaded from DDF descriptions using the "-f" switch as follows

```
        cubix -f domain.ddf
```

This command allocates and loads processors according to the description found in the named file.

4 Creating and Using Domain Descriptions

Even if your system is "homogeneous" you can use the domain description methodology to allocate and load programs. In this case the name of the domain supplied in the DDF file is arbitrary and will, in fact, be ignored by *Express*. This allows you to use the same domain description file on several different homogeneous machines.

Creating domain descriptions.

When creating description files for heterogeneous architectures the first thing you will need to know is the names of the configured domains, and their contents. The simplest way to get this information is with the domain configuration tool, "domtool". Executing the command

```
        domtool -l
```

will produce output similar to that shown in Figure 4. For each known domain the included "machines" are listed together with the number of "programs" that can be run there and a description of whether the "processors" are "attached" or "local". Using this information and knowledge of the various system capabilities one can build appropriate domain descriptions.

```
Domains:
     SUN4:
         sampson     4 local programs.
         pollux      2 local programs.
         delilah     4 local programs.

     SUN3:
         wega        2 local programs.

     SUN386i:
         procyon     1 local program.
         simic       1 local program.

     GRAPHICS:
         kastor      1 local program.
```

Figure 4 A Listing of Available Domains

5 Configuring Domains

Having described the mechanisms by which domains are used in assigning programs to nodes, and nodes to hardware we must discuss the issue of "system configuration" in which the domains are themselves created. The procedure is normally carried out graphically with the domain configuration tool, "`domtool`", and is performed by the system administrator. After setting up certain configuration files the `exinit` command is then used to start-up the necessary network servers which are used for communication between the various "machines". In this sense the process is entirely analogous to the "`cnftool`"/"`exinit`" sequence used to configure other *Express* systems.

`domtool` *plays a similar role to* `cnftool`

Information required to configure domains

In order to build the domains on which you system will operate you need several pieces of information:

- The names of the domains. These can be *any* strings although those that describe underlying system architectures are easiest to use and remember.
- The (network) names of the machines which will be assigned to the domains.
- The number of "programs" that you wish to be able to run on this machine.
- Whether the "processors" are "local" or "attached".

In addition information should be supplied regarding the network interconnectivity required among the machines. This information is used by *Express* to build communication paths between the various machines - it is not a physical description of the underlying hardware. You merely indicate which machines should be directly connected by ethernet sockets and which will communicate by through-routing. *Express* takes care of all the details so that your applications remain unaware of the underlying network protocols and/or connectivity - this information is only required for system configuration.

5.1 An example workstation system

A workstation system

To make the ideas behind domain configuration more definite we will work through an example of the use of "domtool" to build a network consisting of three workstations. We will assume that the following machines are available:

- "gobi", a Sun-4
- "sahara", a Sun-4
- "kalahari", a Sun-3

The trade-offs in assigning machines to domains

Since the machines are not all of the same architecture we cannot construct a system with a single domain. On the other hand we probably don't want to make each node its own domain since that would force the user to always specify the machine on which their programs should execute. (*Express* tries to load balance applications within a domain by assigning nodes to free machines in preference to those which are already in use.)

The best compromise, therefore, is to make two domains. For convenience we choose the names "SUN4" and "SUN3".

The trade-offs in assigning "nodes" to machines

Having made these decisions we must now consider the assignment of "programs". None of the machines has an attached parallel processor so the number of "attached nodes" will be zero. As a result we are free to assign any number of "local nodes" to the machines. For optimum distributed performance we would probably like to assign a single "local node" to each machine. In this way a program that uses 3 nodes would be allocated one on each workstation resulting in best CPU performance. On the other hand our network would then be limited to running only three "nodes" at any one time which may be too restrictive.

In this example we choose to assign three "local nodes" to each of the Sun-4 machines and two to the Sun-3. This yields a total of 8 "local nodes" and also takes advantage of the greater processing power of the Sun-4.

The "system console"

The last issue to be resolved before starting the configuration system is that of the "*Express* system console". One of the workstations should be selected as the system console. While all machines will appear equivalent from the perspective of the user the system console is special in that it is the only machine capable of executing the "exinit" system call which reloads the system software and starts the network servers. On this machine we should now execute the command

```
domtool
```

The special files in Express

When started domtool checks for the existence of certain important system configuration files to see if a previous configuration is available. The names of these files vary from system to system but are maintained in the standard *Express* data-base, the "customization file". The currently used files are represented by the following macros:

NETFILE	This file contains a description of the ethernet interconnections between machines.
CONFILE	Contains the forwarding information that *Express* will use to send messages between nodes.
PLOTFIL	Describes the most recent image of the network as displayed by

```
domtool.
```

(Note that *ParaSoft* reserves the right to alter the contents or names of these files in any future release of *Express*.)

If "`domtool`" detects these files it will ask whether you wish to continue from the existing configuration or to start from scratch. In our case we wish to build a network from scratch so you should answer appropriately. (You may not be asked this question at all if `domtool` doesn't think that a configuration currently exists on your machine.) At this point the display should contain an image such as that shown in Figure 5.

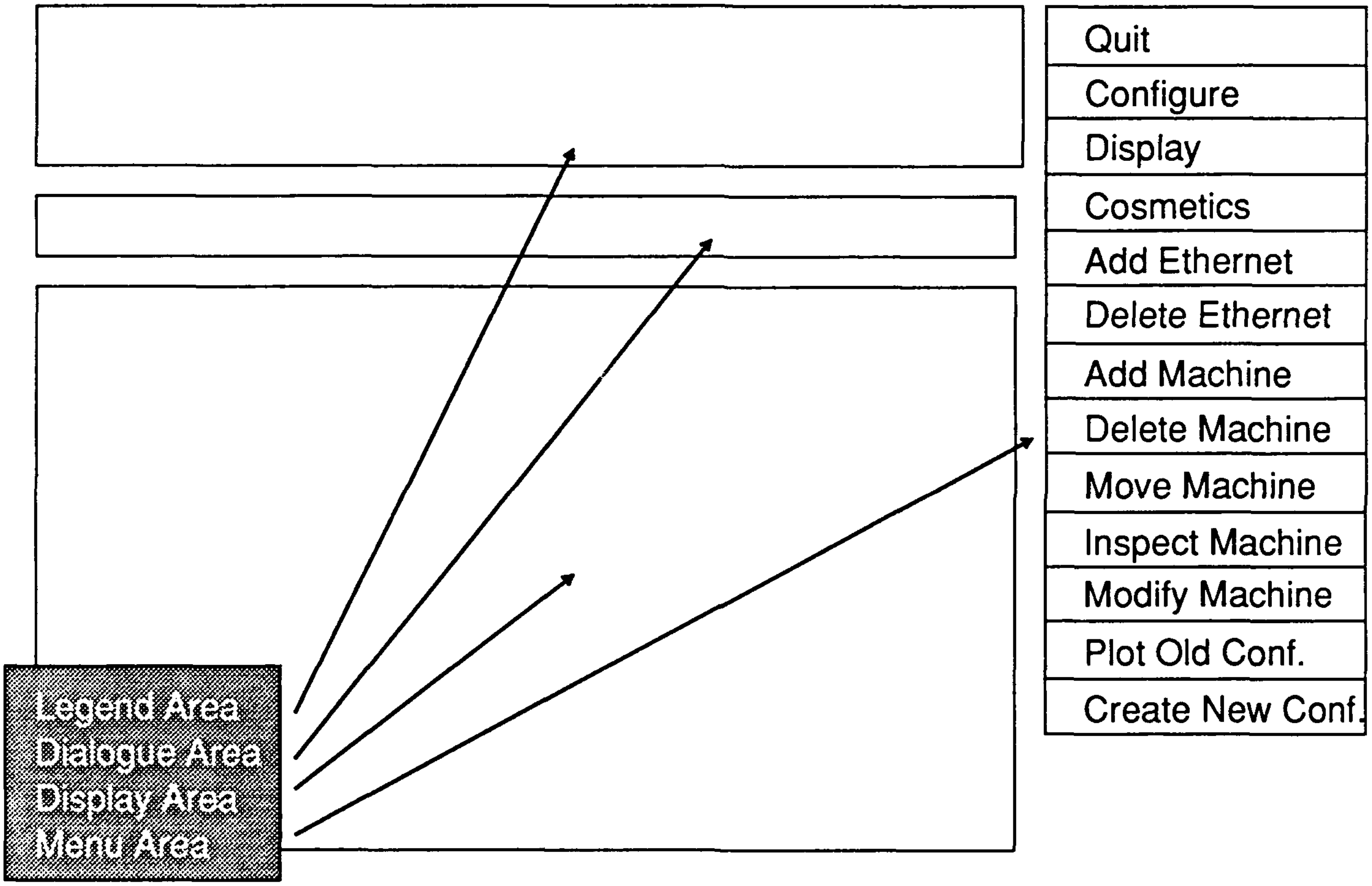


Figure 5 Basic `domtool` display

The top part of the display contains an area which is used to display the color coding used to indicate domains. The right hand side of the display has a menu containing commands which are executed by "clicking". In the main display area the "machines" will be represented by named icons. Ethernet connections between workstations are shown as solid lines. The "Dialogue Area" shown in the Figure is used to indicate instructions and other informative details to the user. While manipulating networks with `domtool` this area will contains details of how to perform certain operations.

Elements of the network description

The network description we will create consists of two elements: machines and ethernet connections. The former represent the CPU resources of our network and are assigned to domains when created. The ethernet connections are required by *Express* when forwarding messages between different machines.

Adding machines to the network

To create the configuration of our system we need to create and assign two Sun-4 and one Sun-3 workstations. This can be done in any order and we choose to create "gobi" first. We add a machine to the network by selecting **Add Machine** from the menu. The dialog area of the display will prompt us to position the icon in the display area. Do this by moving the cursor to the position where the machine's icon should appear and clicking the left button. At this point a workstation icon should appear on the screen and a dialog box will prompt for information about this machine, see Figure 5.

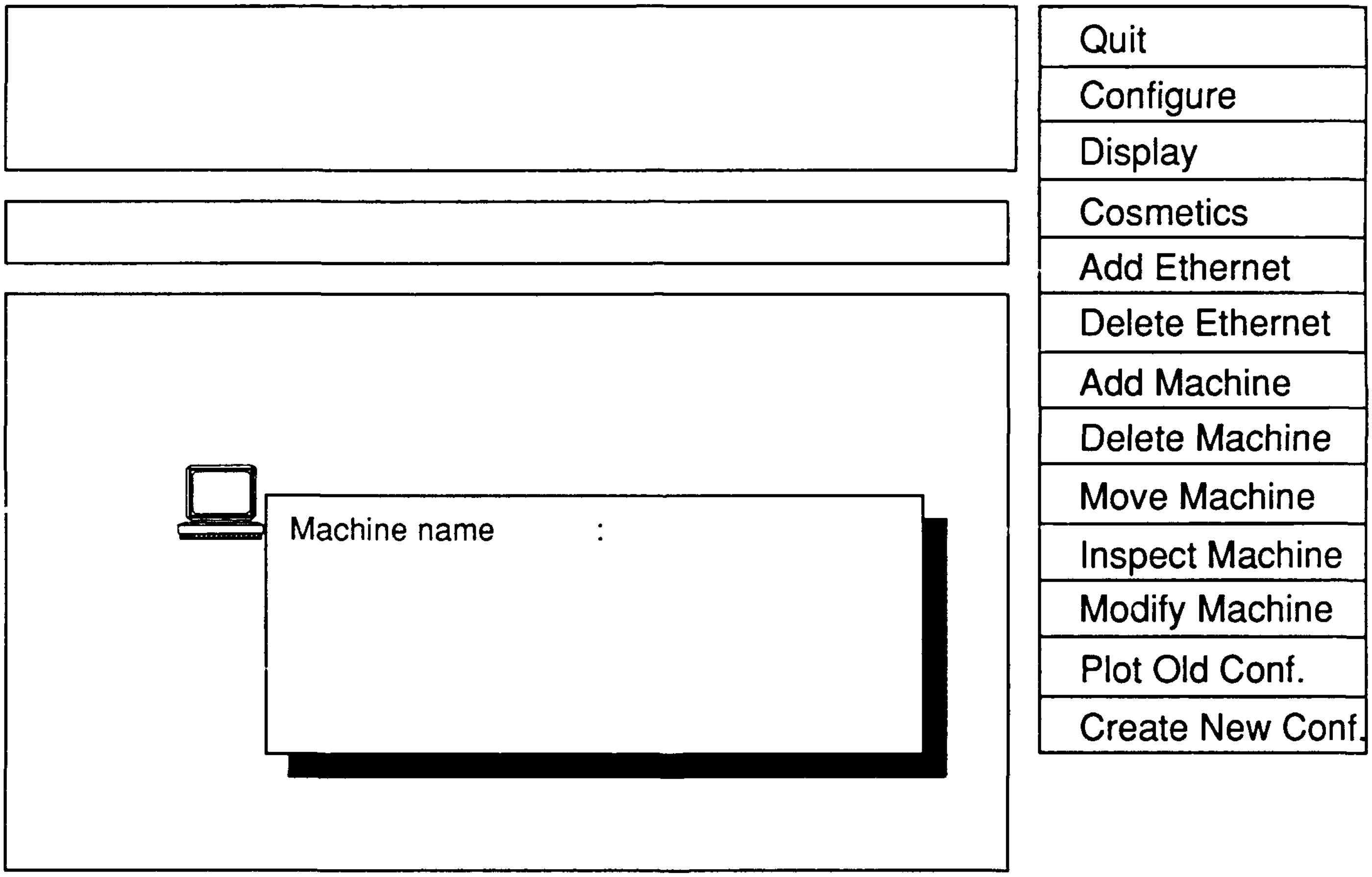
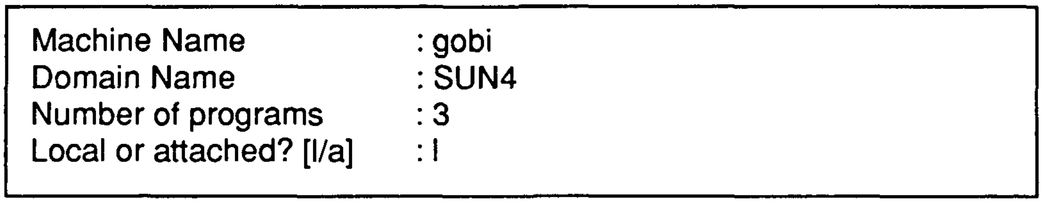


Figure 6 Adding a new Machine

Supplying information about domains and nodes

We are being prompted for the "name" of the new machine. This information is used when booting the *Express* system over the network and should be the "network" name of the machine. In this case we would answer "gobi". In turn we will then be prompted for the "domain" to which this machine should be assigned, how many nodes this machine has and whether they are "local" or "attached". In our case we have already determined our answers and the completed “Data Area” should appear as shown in Figure 7.



```
Machine Name               : gobi
Domain Name                : SUN4
Number of programs         : 3
Local or attached? [l/a]   : l
```

Figure 7 The Completed data area.

After completing the information the machine’s icon in the “Display Area” will be changed by the addition of the name supplied in answer to the first prompt. A new icon should also appear in the “Legend” area showing the domain to which the new machine belongs. This will aid you in designing the overall network. Adding the other two machines is similarly achieved by selecting the **Add Machine** option from the menu, the only trick being to remember to use the domain name “`SUN3`” for machine “`kalahari`”.

After adding all three machines the display will probably look somewhat similar to

Figure 7. At this point the assignment of "machines" to "domains" is complete.

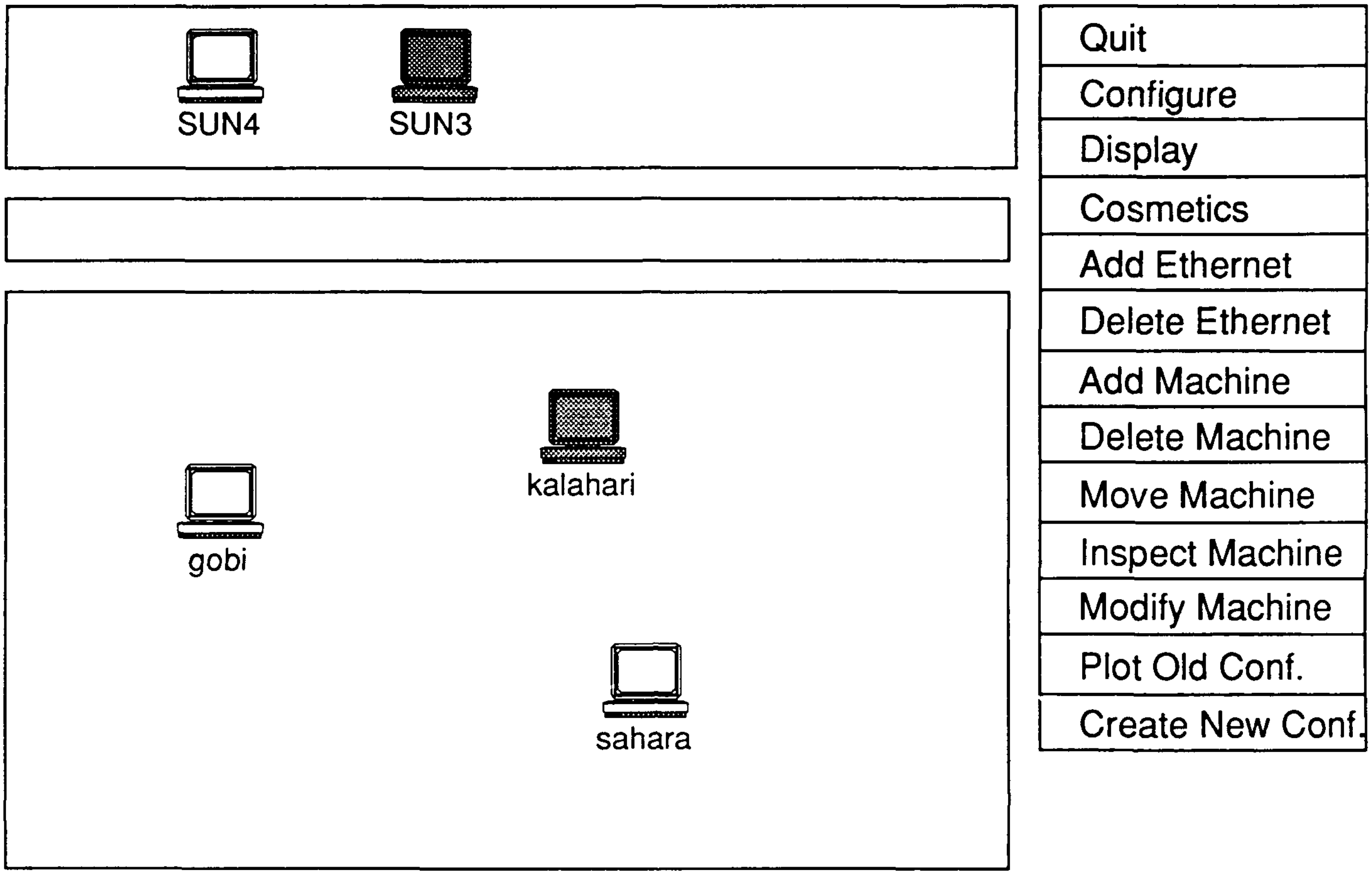


Figure 8 After adding all three machines

Note how the various domains are shaded. This allows you to quickly see which machines belong to which domains.

Assigning Ethernet connections

Before *Express* can be fully configured, however, you need to "connect" the machines in the "Display Area" with ethernet sockets. To do this we use the **Add Ethernet** option from the main menu. You will be p:ompted, in the "Dialogue Box", to select a pair of machines with the mouse. Having selected an pair of machines the ethernet connection will be indicated with a line, as shown in Figure 9. If you have trouble selecting objects it is

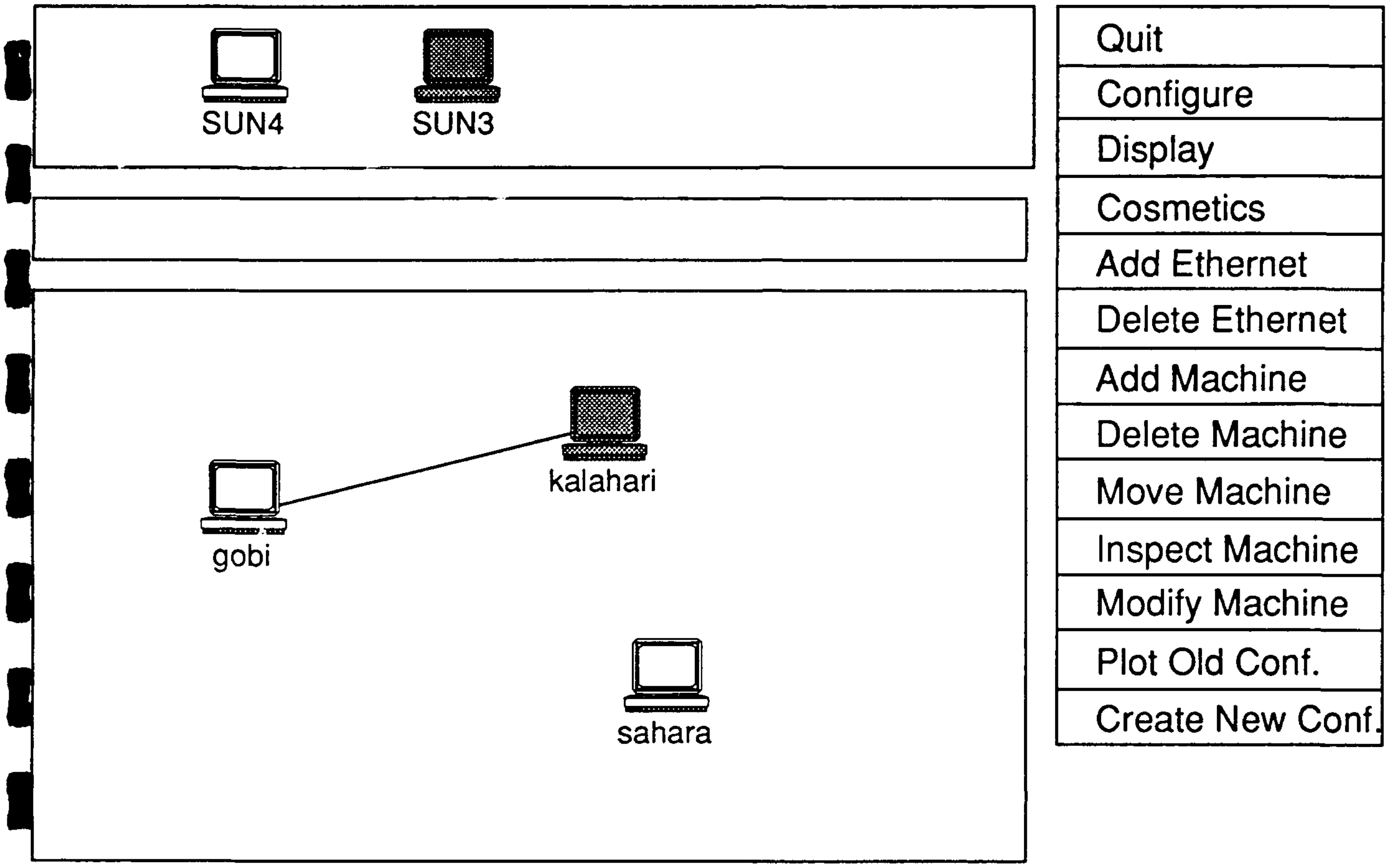


Figure 9 Adding an Ethernet Connection between "`gobi`" and "`kalahari`"

probably because your "mouse clicks" are not near enough to a machine to be recognized as identifying that workstation. In this case you may see a message telling you to "exit" by clicking with the right button, or double-clicking something. In this case just keep trying.

Note that these connections do not refer to a physical hardware link between machines, but will be used to set up Ethernet "socket" connections between machines when `exinit` is executed. These links will be used internally by *Express* to forward messages between "nodes" attached to different "machines". This process is quite transparent to the user who can program as though every node were connected to every other. By repeating the process by which "`gobi`" and "`kalahari`" were connected we can add two more links to create

Indicated "connections" need not by "physical" wires.

the display shown in Figure 10. At this point every machine has a direct connection to each

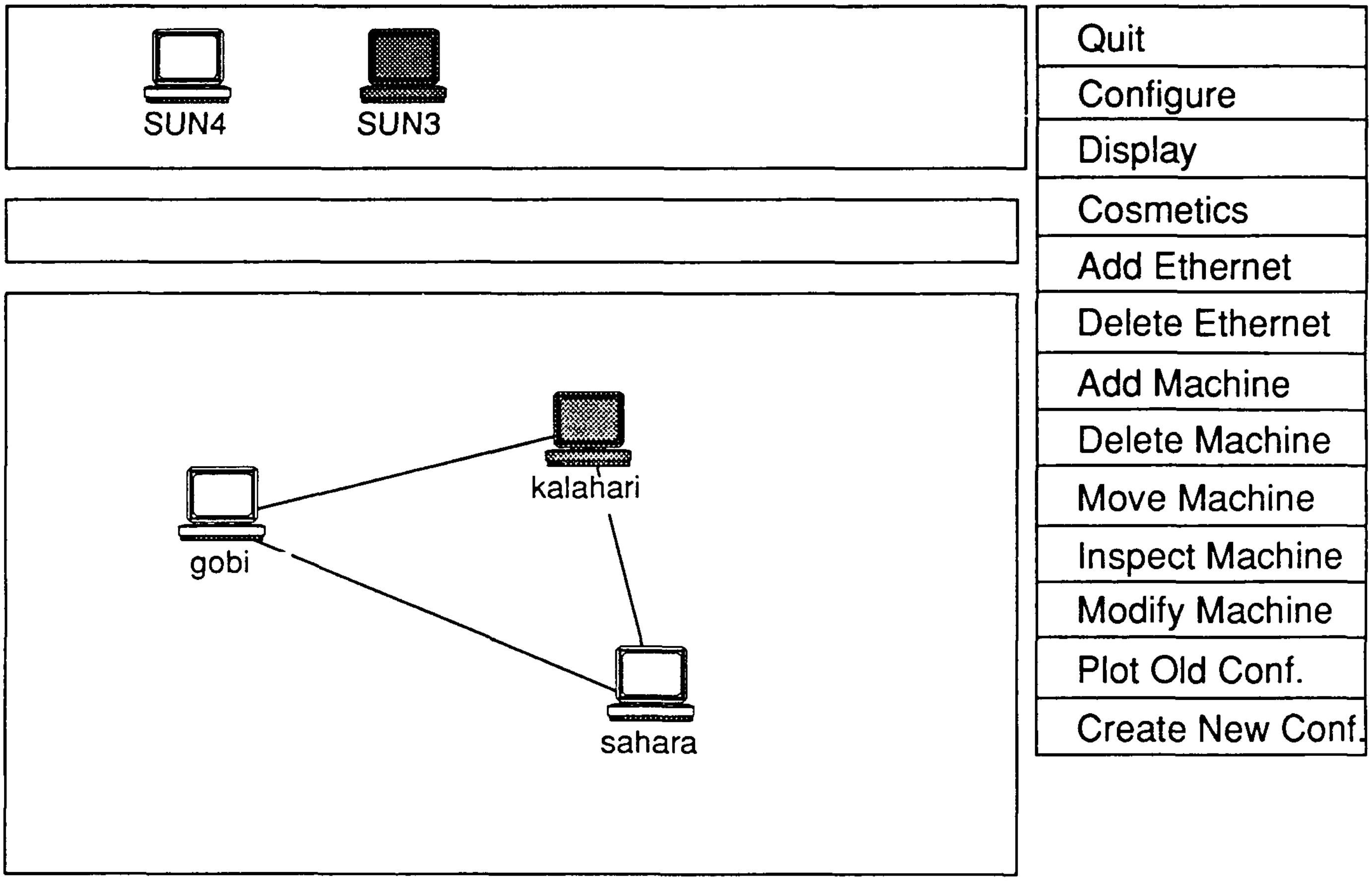


Figure 10 A Full Interconnect

other machine and adding more connections is redundant.

Trade-offs in allocating ethernet connections

It is important to note that network resources such as the “socket” links used by *Express* are not necessarily cheap on all systems and may incur overheads of their own. As a result it is *not* always beneficial to build a fully interconnected network such as that used here. While this is feasible for three machines, using only three socket pairs, a fully connected network of 16 machines would require 120 sockets which is rather more than can be maintained on most systems. The solution in this case is to use less direct connections and to let *Express* to perform forwarding between machines which are not directly connected. This action is transparent to the user application. In practice we find that four connections per workstation is a reasonable maximum.

Saving the configuration

Before exiting from `domtool` we must create the configuration tables for the system and save them in the places indicated by the “customization file”. This is achieved by selecting **Configure** from the main menu. If you see the message

```
Configuration Failed - topology is not connected
```

this means that the network you have shown consists of two or more disconnected pieces - there are some machines which *cannot* be connected to others, even by forwarding through intermediates. In this case you need to add more ethernet connections to your system with the **Add Ethernet** option.

5.2 Cosmetic Improvements

When building the network description you will occasionally find that the display becomes cluttered. In these cases the **Display** and **Cosmetics** menu options are available to rectify matters.

The **Display** menu offers options to scroll the display in various directions and completely erase and redraw the displayed system.

The **Cosmetics** selection offers menu items which can be used to enhance the displayed image in several ways.

The most important type of cosmetic improvement involves adding "vertex" points to the connection lines. The idea of this option is to allow "bends" in the lines representing ethernet connections. To try it out select the **Make vertex** option from the **Cosmetics** menu. You will be prompted to indicate a line which should be "bent" and a location for the "vertex". The result of selecting the link between "`gobi`" and "`kalahari`" and a point roughly midway along and above the original line results in an image similar to that of Figure 10.

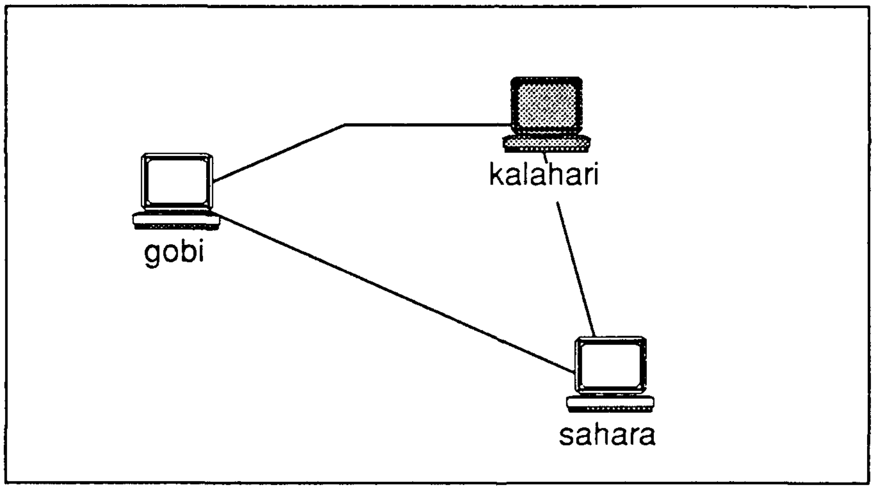


Figure 11 Result of using the "Make vertex" command

Using this technique it should be possible to create a "clean" representation of the network.

6 `domtool` without graphics

It is easiest to create network configurations using the graphical interface just described. If, however, you are unable to execute the graphical interface on your system an alternative, line oriented, interface is also available.

Things to do before starting `domtool`

To show the use of this system we will build the same network as shown in Figure 10, using the more primitive interface.

To configure a system without graphics you will find life much simpler if you create, on paper, an image similar to those already presented. In addition you should add a number to each icon, starting with 0. For our example we might select the numbering

0	gobi
1	kalahari
2	sahara

To use the line-oriented interface to `domtool` type the command

```
domtool -p
```

As in the graphical case you may be asked whether you want to proceed from an existing configuration or begin again. It is normally simplest, in the absence of graphics, to start afresh.

Initially you will be asked to indicate the number of machines in your network. In our case there will be three machines so we respond

```
3 Return
```

(Note that the "`Return`" in the above should not be entered as text - you should press the key marked "`Return`", or "`Enter`".)

Next you will be prompted to enter specific information for each machine in the network, in order of increasing "number". This is the reason that you should number the machines in your sketch plan before starting `domtool`. The basic information required is in the same format as shown in Figure 7 except that you will also be asked to enter information about the ethernet connections to be made to this node. You will see the prompt

```
Please enter up to # ethernet connections:
Machine 0: Connected to ? (-1 to quit) :
```

The exact number of links allowed is preconfigured and may vary from implementation to implementation. It is unlikely to be less than four. In any case you should enter the "numbers" of the machines to which this one is connected, finally giving -1 when all connections have been made. In our case "`gobi`" (number 0) is connected to both "`kalahari`" (number 1) and "`sahara`" (number 2) so we would enter

```
1 Return
2 Return
-1 Return
```

i.e., link 0 goes to the machine 1, link 1 goes to machine 2.

We have now described everything necessary about machine 0, "`gobi`".

`domtool` will now prompt you to enter similar data about the other machines in the network. Rather than go through the laborious details of explaining the various responses we merely present the correct answers. Hopefully their meaning will be clear.

```
Machine 1:   kalahari
             SUN3
             2
             1
```

```
                  0 2 -1
      Machine 2:  sahara
                  SUN4
                  3
                  1
                  0 1 -1
```

7 Booting *Express*

At this point we should have built the appropriate configuration files and have them saved in their correct locations. Basically this process entails the creation of two special files called "netfile" and "confile" which contain information about the way in which the domains are configured and the manned in which network connections can be created and utilized. At this point we should be able to re-load the *Express* system with the exinit command.

Running exini ***on the network system.***

If all goes well you should see the following response

```
ParaSoft Network Configuration
==============================
Killing existing servers...
Booting new servers
            gobi
            kalahari
            sahara
Done
```

If there are servers currently running they will either be killed, or exinit will fail depending on the options used to execute exinit. As each machine is initialized its name is displayed. Finally, after all ethernet connections have been established exinit terminates and the system is ready for use as described in Section 3.