4-20-1994

Developing a Benchmark for Evaluating the Performance of Parallel Computers

D. Ladd Williamson
Utah State University

Developing a Benchmark for Evaluating the Performance of Parallel Computers

D. Ladd Williamson

April 20, 1993/4

EE492 Technical Report

Utah State University
Electrical Engineering Department

and

Argonne National Laboratory
Mathematics and Computer Science Division

Abstract

This paper discusses the development of a portable suite of benchmarking programs for parallel computers. Comparative measurement of the performance of parallel computing systems has been limited because of the great diversity of architectures and of processor interconnection schemes. One solution is to translate benchmark codes into a consistent and portable parallel language. This paper reports on progress in developing such a portable suite of benchmarks. An extensive introduction to parallel computing is included as an appendix, to provide a thorough understanding of the factors complicating development of the performance suite. Key to the development was the use of p4, a library of tools developed at Argonne National Laboratory. The benchmark codes were translated successfully using p4 and were run on a variety of parallel machines. Conclusions and suggestions for future work are given.

Contents

List of Figures

List of Tables

1 Introduction

Parallel computers have enabled researchers to attack larger and increasingly more complex problems. At the same time, the diversity of parallel architectures has made it important to know which computer is best suited for a particular problem class. Unfortunately, no standard benchmarks exist to compare the performance of parallel computers.

1.1 Project Goals

The objective of this performance evaluation project at Argonne National Laboratory is to create a suitable benchmark for parallel computers. The goal is to standardize existing programs, thus allowing identical source code to be compiled and run on different parallel machines.

1.2 Approach

Several programs exist that could serve as a suitable benchmark across architectural boundaries—if the programs were made portable. At Argonne, the focus was on the Perfect-Seismic codes, a subset of the Perfect-II codes that performs seismic processing. The goal of the original developers of Perfect-II was to provide support for three models of parallel processing: Loop-level parallelism, Fortran 90, and message passing. To date, however, only sequential programs written in Fortran 77 have been completed. My task was to translate these programs into a consistent parallel language based on message passing.

The word *translate* is important here. The intent of this project was not to create new codes but to parallelize and standardize existing software.

Key to this work was the use of `p4`, a library of macros and subroutines developed at Argonne for programming parallel machines. With `p4`, I prepared parallel versions

of the codes on a network of workstations and then ported the codes to several parallel machines.

1.3 Scope

Perfect-II comprises three applications: SEIS (pre-stack seismic processing), FD-MOD (3D finite-difference modeling), and FKMIG (3D Fourier domain migration). Because FKMIG is already being parallelized by other researchers, the project at Argonne focused on SEIS and FDMOD.

The target machines used for the parallel work were as follows:

Intel iPSC/860 A local-memory, 8 node hypercube at Argonne. A 64 node iPSC/860 at Caltech is also available over the Internet.

Intel DELTA, with 512, nodes at Caltech. The DELTA is connected in a mesh and capable of scaling to 2048 nodes.

Sequent Symmetry shared-memory multiprocessor at Argonne, with 26 nodes.

BBN TC-2000 shared-memory multiprocessor, with 45 nodes, at Argonne. Memory is local to individual nodes but is treated as shared. Remote access to memory can be accomplished via a butterfly switch.

Alliant FX/2800 shared-memory, 28-node multiprocessor at Argonne. Their Fortran 77 compiler is instructed to parallelize standard code, automatically looking for loop parallelizations.

Kendall Square KSR-1, 64-node multiprocessor at Oak Ridge National Laboratory. Memory is physically distributed but treated as shared. The nodes are connected in stacked 32-node rings.

Thinking Machines CM-5, with 512 nodes, at University of Illinois Urbana-Champaign. In this local memory machine, nodes are connected by a fat-tree topology where the bandwidth is the greatest at the root of the tree.

nCube 6400, with 192 nodes, at MIT. This is a local memory machine in a hypercube topology.

IBM SP-1 with 32 nodes at Argonne. This machine is a local memory architecture, with each node containing the same processor as their RS6000 workstations and 128MB of RAM.

This list comprises the major participants in parallel supercomputing. The architectures include shared-memory machines, local-memory machines, and hybrids (such as the KSR-1, which physically has a distributed memory but acts as a shared-memory machine). The interconnection scheme describes how nodes, memory, and I/O modules are connected. In this group are found shared buses, hypercubes, mesh topologies, and butterfly switches. Given the fundamental differences in architecture, running the same program on each platform represented a major challenge.

1.4 Plan

This report is organized as follows. Section 2 gives a brief review of parallel processing. Section 3 discusses p4 and the visualization tools that facilitated parallel program development. Section 4 presents a brief look at benchmarking in general and the Perfect benchmark suite in particular. Section 5 details the work on the two codes FDMOD and SEIS performed at Argonne; included are some of the problems encountered and results obtained. Also discussed is future work needed before this parallel benchmark is fully usable. Finally, Section 6 draws conclusions about the project and the benchmarking of parallel computers. Appendix A provides a much

more lengthy review of parallel processing, to illustrate the great diversity of parallel architectures, network configuration, and program models and, thus, underscore the challenges to developing a standard, portable benchmark suite. Appendix B contains the Argonne Technical Report written by the author which will be distributed with the actual benchmarking codes.

2 Parallel Computers

While all uniprocessors function under the von Neumann architecture (fetch, decode, execute) there exists no underlying standard for parallel machines. As manufacturers have taken different hardware approaches to parallel processing, the fundamental workings of varying multiprocessors can be quite different. A very brief overview of parallel processing will be given here.

Of today's multiprocessers, most fit into one of two broad classes depending on the mode of communication between processors. Shared memory machines consist of processors and a single common memory. Communication occurs via shared variables, with each processor accessing the shared variable when information is needed from another processor. While programming shared memory multiprocessors is relatively straight foreward and syncronization and communication costs are relatively low, problems arise when more than one processor tries to access a shared variable.

Local memory multiprocessors have no shared memory, but rather a private local memory attached to each processor. Communication then takes place through explicit message passing. This requires carefull programming and is very slow. However local memory machines are much more economical and scale more easily, hence most new machines are of the local memory architecture.

Because of the low overhead of communication, shared memory machines tend to exploit fine-grain parallelism. This would be represented by sending independent

statements in a function to different processors. The time required to send messages in a local memory machine encourages coarse-grain parallelism. Commonly whole programs are run in parallel, with each processor having its own copy of the program and communicating only rarely with another processor.

Many different methods have been used to physically connect processors, I/O processors, and memory. Among these are the Shared Bus, which uses a single communication path between nodes, Crossbar-Switch Matrix, in which each node is connected to every other, and the Hypercube, where large numbers of processors are connected relatively cheaply.

Since very different programming models are used for different multiprocessors, no single programs have been run on different classes of parallel computers. Benchmarking has been difficult and inaccurate, and no guidelines have been available on which to base conclusions as to the merit of one system over another.

3 p4

The diversity of machines and of parallel programming models motivated the development of the p4 system at Argonne National Laboratory in the 1980's. The p4 system is a library of macros and subroutines for programming parallel machines. It grew out of the effort described in the book *Portable Programs for Parallel Processors* [4], from which p4 takes its name. One may think of p4 as an extension to C or Fortran, since it consists primarily of C functions that achieve process communication and synchronization. Because a Fortran interface is provided, C and Fortran programmers need not learn a completely new language.

3.1 Portability

An important virtue of p4 is portability. It started out as simple macros used to change parameters in function calls, depending upon the machine for which the

program was being compiled. Since many message-passing primitives work in a similar fashion and all that might be required is the inversion of some function parameters, this worked well. Now, however, this has grown into what must be deemed as a new language. The authors determined what philosophy should be used in programming parallel machines and then wrote the abstract p4 function calls required to accomplish it. Because portability was an original motivating factor, support for many new architectures were included; currently, 24 platforms are supported.

The portability of a p4 program is easily achieved. Specifically, any p4 program that runs on one architecture requires only a recompilation on a new architecture to run in the new environment. Of course, various machine-specific flags must be set at compile time; but the source code can be used without modification.

The p4 library frees programmers from the specific details of parallel programming. By using abstracted function calls, the details of the interface to the machine may be ignored.

3.2 Models Supported

The p4 system supports three programming models: Shared memory, message passing, and cluster programs. To develop programs in a shared-memory environment, p4 had to develop a scheme to protect shared variables. This is done by use of *monitors*. A monitor protects shared modifiable data by allowing only one processor at a time access to that data. A look at the specific monitors provided by p4 will evidence the success of abstracting communication constructs from the hardware requirements to a logical level. A simple `LOCK` monitor may be used to protect a critical section of code containing a shared modifiable variable. The `BARRIER` monitor forces some number of processors to wait until all of them have reached a certain point. This represents a simple way to achieve process synchronization. The `GETSUB` monitor is used to obtain the next value of a shared counter (the next subscript for an

index variable). The ASKFOR monitor functions like a general dispatcher of work. It requests a new "problem" to work on from the problem pool. With this flexible array of functions to manage shared variables, efficient shared memory programs may be written in p4.

The p4 system supports a set of send/receive procedures to accomplish message passing. These procedures are "generic" in the sense that they do not know whether a message must travel across a network or through shared memory or via some other mechanism. When a processor requires a reference to a variable not located in its own memory, it sends a message to the owning process. That processor will fetch the data and then send a message back with the requested data. Also, p4 provides several mechanisms for dealing with all processes at once, such as a broadcast of information that goes to all processors.

Special cluster management functions are included in p4. Once a cluster is organized, regular monitors are used for the processes within that cluster. One process per cluster will be designated the cluster master and will be responsible for all inter-cluster communication. Normal p4 message-passing routines are used for this message passing, with the restriction being that only cluster masters may send or receive messages.

3.3 Alog

Distributed with p4 is a set of routines for creating logfiles, called Alog. The created logfiles consist of time-stamped events. The timestamps are obtained from various microsecond-level resolution timers, depending on the specific machine.

To use Alog, users simply need to call the Alog initialization functions before any event is logged and then define the type of event to be logged. The only step left is to place a logging call at important areas of execution.

On networks of workstations and some distributed memory machines, the microsecond timers on the various processors are synchronized. To produce a usable merged logfile, a program has been included to adjust the timestamps for offset and drift before they are merged.

3.4 Upshot

While not included with the distribution of p4, Upshot may be freely obtained for use with p4. Upshot was designed to examine Alog produced logfiles and to produce a visual representation of parallel program behavior. To increase the usefulness of such data, the user may define certain events to mark entrance and exit into *states*. With the combination of Alog and Upshot, programs may be more easily understood; and new insight may be gained as to the workings of a particular program. Figure 1 shows a typical view of upshot, with the states clearly discernible by different patterns.

4 Benchmarking

The discussion of parallel machines in Section 2 raises several questions: Which machine gives the best performance? Does performance depend on the class of application? What types of problems scale easily? The answers to these questions depend on the availability of accurate benchmarks.

Benchmarks are programs that measure the performance of computers. When performance is measured, judgments as to the effectiveness of various architectures and configurations may be made. Kuck and Sameh [1] identify three basic purposes for using benchmarks to evaluate existing machines:

1. Selecting a new machine,

2. Tuning of an existing system, and

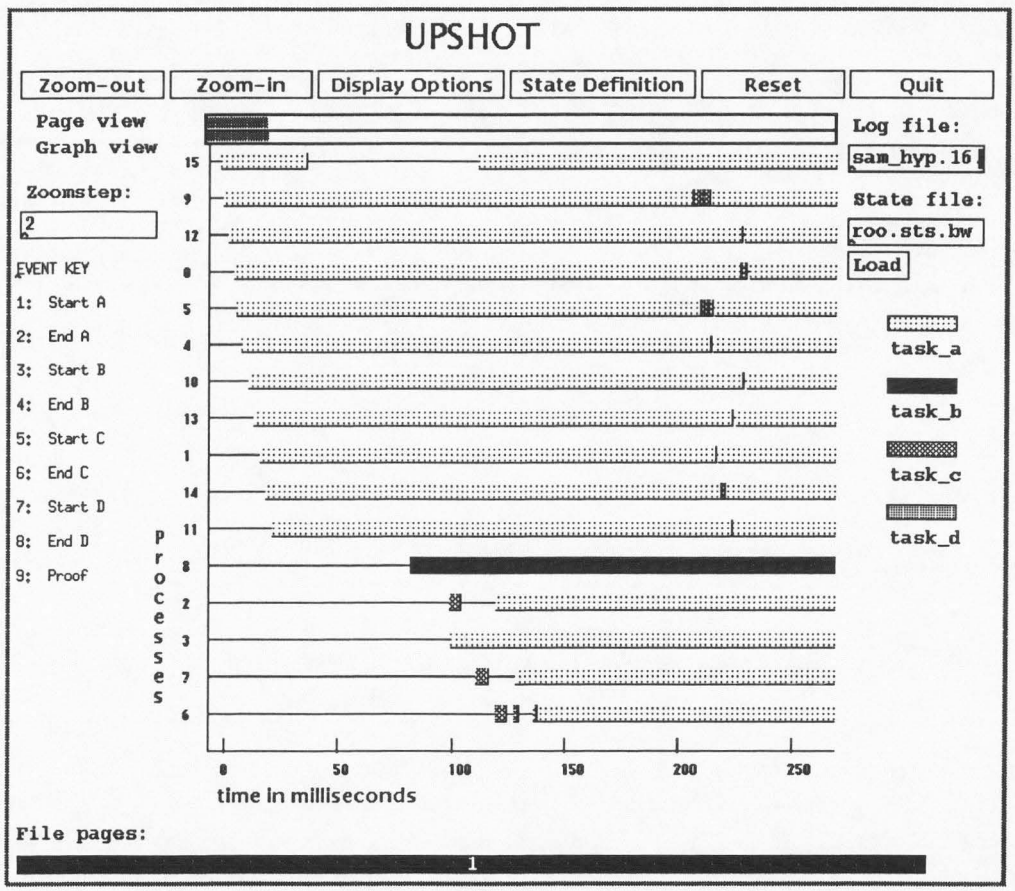

Figure 1: Upshot View of Log File Data

3. Determining which strengths to incorporate and which weaknesses to avoid in the design of a new machine.

Benchmarks can be divided into two categories: Kernel and application. Kernel benchmarks, by far the older and more widely used, employ a small set of computationally expensive loops to compare machines. Application benchmarks compare machines by running whole applications, not just fragments of code. Since application benchmarks run complete programs, they give more reliable performance estimates.

Unfortunately, while standard benchmarks exist for sequential processors, such as the *LINPACK* [3] libraries developed at Argonne National Laboratory and used for measuring floating-point performance, no comparable standard has emerged for parallel processors. This situation makes performance comparisons very difficult, because different parallel architectures are believed to be suited for different problems. A suite of parallel programs that is easily portable to a wide selection of architectures is needed. These machines may then be compared, using the same standard.

4.1 Perfect-I

Motivated by this need and guided by a proposal of Kuck and Sameh [1] to create a benchmarking suite, academic and industrial collaborators initiated, in 1987, the Perfect (**Per**formance **E**valuation for **C**ost-Effective **T**ransformations) benchmarking group. The goal was to produce a large number of applications that could be used in performance evaluation and could be ported to a large number of high-performance computing machines [2].

This effort culminated in 13 programs representing a variety of scientific and engineering problems. The codes were successfully ported to over 30 machines. This effort proved to be the first systematic solution to the problem of benchmarking parallel machines.

4.2 Perfect-II

Supercomputers have matured tremendously in the years since 1987, when the Perfect-I benchmark was being developed. The relatively smallness of the datasets and other problems made it less useful for benchmarking high-end parallel machines. The Perfect-II was introduced to provide a better benchmarking suite. Perfect-Seismic, a subset of Perfect-II with all codes relating to seismic processing, was the first to actually be included in Perfect-II.

The Perfect-II suite contains three applications. The first, **SEIS**, performs prestack seismic processing. An originating process reads seismic traces, which are then piped through a chain of data processing routines. A final process writes the processed traces out to disk or tape. The initial release of **SEIS** includes the following processes [3]:

DCON – seismic trace deconvolution
DGEN – synthetic data generation
DMOC – dip moveout correction
FANF – 2D spatial filtering by Fourier transform
GEOM – seismic geometry specification
NMOC – normal moveout correction
READ – read seismic benchmark file
RTMG – reverse time finite difference migration
STAK – stack seismic traces
WRIT – write seismic benchmark file

A processing flow would include process "READ" to input the initial parameters, one or more of the processing routines, and then process "WRIT" to capture the output.

The second program, **FDMOD**, performs 3D finite difference modeling. This forward modeling is accomplished with the acoustic (scalar) wave equation.

The final program, **FKMIG**, accomplishes 3D Fourier domain migration. Multi-dimensional Fourier transforms are used to obtain images of geologic structure from recordings of pressure on the surface of the earth.

As noted in Section 1.2, the initial release of Perfect-II was supposed to support three models of parallel processing: Loop-level parallelism, Fortran 90, and message passing. However, only sequential programs written in Fortran 77, suitable for automatic loop parallelization by the specific compilers, were included. Neither Fortran 90 array syntax nor message passing was supplied.

Two institutions have recently undertaken an effort to complete the work on Perfect-Seismic. A group at Dartmouth College is translating **FKMIG**, and Argonne is translating **FDMOD** and **SEIS**.

5 The Argonne Project

The project involved several steps:

1. Learn how to use p4 for translating from standard Fortran 77.
2. Master the Alog and Upshot visualization tools provided by p4 for displaying the parallel behavior of codes.
3. Translate **FDMOD** and **SEIS** into p4.
4. Port each program to eight parallel machines.
5. Document the results.

The first two steps required writing simple programs in C (since p4 is written in C with a Fortran 77 interface), experimenting with the various logging features of p4, and studying the users manuals for the Perfect-Seismic benchmarks. The remaining steps were done separately for the two codes, **FDMOD** and **SEIS**.

5.1 FDMOD

FDMOD was selected to be first because the author had already written a parallel version that used the message-passing functions specific to the Intel machine.

Initial attempts to run the parallel code on the Argonne iPSC/860, however, uncovered the fact that the code had many bugs. These ranged from simple errors, such as the wrong number of parameters passed to a subroutine, to very subtle errors, such as having two similar variables transposed in a formula. Over a week passed before the (supposedly) correct code was running. The rough structure, coupled with the many errors, made this version seem to be an unfortunate basis for creating a new message-passing version in p4.

Fortunately, Argonne was able to obtain a message-passing version being developed at MIT for the nCUBE machine, with the message passing being organized either through nCUBE calls or what were called p4 wrappers. These wrappers were actually functions that used p4 to accomplish nCUBE calls. This version was, of course, not portable; but it was closer to what could be set up as a message-passing standard, and the code was much cleaner.

Translating the MIT code to p4 proved to be much easier than working on the author's parallel version for two reasons. First, the code was written in a much more modular fashion, with the program flowing more naturally. Second, p4 had been used in the code already. Again, a complication did arise: p4 had been used to rewrite the nCUBE calls, contrary to its intended purpose; nevertheless, the effort now focused on fixing misused p4 calls rather than porting to a totally new language.

With the code successfully running on a network of workstations, the port to different architectures began. Specifically, this involved (1) taking an exact copy of all files from the network to the target architecture, (2) setting the machine-specific compile and linking flags correctly and (3) learning the procedure for running programs on each machine.

The work on **FDMOD** proceeded relatively smoothly. The same source code was used successfully on the Intel DELTA, the Intel iPSC/860, the Sequent Symmetry, the BBN TC-2000, the Alliant FX/2800, the nCUBE 6400, Kendall Square Research's KSR-1, IBM's SP-1, and IBM's RS6000 and Sun's Sparc workstations. The CM-5 has proven difficult and has not yet been run.

Even without a successful run on the CM-5, however, this work shows the remarkable portability of p4. To provide a consistent measuring stick, the same benchmark program can be run on widely varying architectures. Although the focus has been to provide the basis and method for parallel benchmarking, actual timed runs have been made; and the results are summarized in Table 1. The row headings are the different architectures timing runs were attempted on. When a machine might use either shared memory or message passing (unix sockets) the specific memory access methods is specified. The column headings give the number of processors and the size of the problem. For example, the second column tested two processors with a problem size of 70 test points per side for a cube. The times are in seconds.

5.2 SEIS

SEIS performs many processes, which may be run in varying order, on a stream of seismic traces. The code is far more complex than **FDMOD**, with many more subroutines used and a separate directory of utility functions.

The author of this code, Charles Moser, had constructed a message-passing parallel version that is specific to the Intel machines. To help manage such a large project,

Table 1: FDMOD timing results

Machine	1 (50)	2 (70)	4 (80)	8 (110)	16 (130)
Sun	3.08019	7.63494	7.20896	30.8019	
nCUBE	4.25984	5.76716	4.32537	8.65075	6.02931
iPSC/860	1.83700	2.30100	1.75500	2.36000	
DELTA	1.64100	2.21900	1.69300	2.35100	2.23900
KSR - SM	2.73999	3.96000	3.31999	4.29999	4.01999
TC-2000 - SM	28.785	75.536	86.546	104.41	103.019
TC-2000 - Sock	28.785	97.297	115.75	392.09	1050.85
FX/2800 - SM	4.56	5.68	4.95	8.51	8.07
FX/2800 - Sock	4.56	6.98	8.97		
Symmetry - Sock	42.86	116.050			
CM-5					
RS6000	3.96799	7.03999	12.2880	34.1759	
SP-1	.639999	1.15199	2.55999	31.2320	42.3680

he put the separate processes into two library archives: One for routines specific to **SEIS** and one for more general seismic-processing subroutines. The rules to compile and link **SEIS** are very general but include a separate file for incorporating machine-specific options. This approach simplifies the code by keeping all machine-specific variables in one place.

Nevertheless, even with this careful management, **SEIS** proved difficult to translate to p4. Not only was the code very large, but several C functions had been included. These provided C functionality but introduced many non-standard Fortran 77 programming practices. The first task, then, required eliminating these C functions from the code. This was done, with the exception of one C function.

SEIS requires many machine-specific capabilities, such as archiving and retrieving libraries. Because of the extra time required to complete each move to a new architecture, ports were carried out only to a select subset of parallel machines. In particular, since Sequent, BBN, and Alliant are all out of the scientific computing arena (BBN and Alliant are out of business, and Sequent now works only with commercial parallel computing), ports to those platforms were not attempted.

Table 2: SEIS timing results

Machine	1 (S1)	1 (S2)	2 (S1)	2 (S2)	4 (S1)	4 (S2)	8 (S1)	8 (S2)
Sun	1.20.1	41.47.1	57.9	43.02.8	0.47.8	18.47.0	0.47.9	10.01.1
iPSC/860	1.36.9	5.12.5	1.31.9	10.45.5	2.41.9	8.12.1		
DELTA	1.29.5	4.14.0	1.29.8	3.40.8	0.46.0	4.42.8	1.02.2	4.45.0
RS6000	2.42.0	2.14.4	1.20.8	1.11.9	1.15.1	1.26.6	0.55.5	1.40.2
SP-1	0.26.6	0.20.9	0.20.1	0.19.4	0.27.8	0.24.0	0.32.8	0.43.7

To date, **SEIS** is running on networks of both Sparc and RS6000 workstations, the Intel iPSC/860, the Intel DELTA, and IBM's SP-1. Efforts are being made to test the newest version of `p4` to use in the port to the nCUBE, KSR-1, and FX/2800. These timing results may be examined in Table 2. The column headings give the number of processors and which of the two example data sets was used (S1 or S2). Times are reported in seconds.

5.3 Future Plans

With some careful work on **SEIS**, it may be possible to remove the final C function. This would standardize the code considerably by removing a function that accomplished a non-Fortran 77 operation. A script to run on each architecture would prove helpful to anyone using the code.

Documentation is also essential. Currently, a document is being prepared that explains the methods used in the new `p4` versions of the code. It contains brief explanations of both **SEIS** and **FDMOD**, the directory structure required, and notes for running on each of the architectures used. This document is targeted at someone unfamiliar with the Perfect-II codes who wishes to run the message-passing suite.

Finally, the work on **FKMIG** being done at Dartmouth must be incorporated into the Argonne work; and the message-passing versions will then be standardized and portable.

6 Conclusions

The work at Argonne marks an *important step* in the development of effective portable benchmarks for parallel systems. This work was aided considerably by the use of `p4`.

Nevertheless, the Perfect-II benchmark effort is just a *first step*. If parallel computers are to reach the processing speeds predicted, much more work needs to be done in the area of performance evaluation. Without a way to judge relative performance by different architectures, it is difficult to suggest realistic design improvements or to guide application programmers in the use of specific machines for specific problem classes. Only through careful notice of the merits of differing hardware and software models can systems designers and programmers learn to exploit more powerful systems.

Replace this with the vitae.

A Parallel Computers

Parallel processing requires two or more processors, each capable of completing tasks independently and concurrently. Many advantages spur this revolution onward. First, parallel processing is a more natural way of solving some problems than is traditional sequential processing. In order to achieve very high processing speeds, it may be cheaper to link together several relatively slow processors than to invest in ultra-high speed uniprocessors. Fault tolerance is increased. ARPA (Advanced Research Projects Agency) has funded the development of several parallel machines and languages through the Strategic Computing Initiative (SCI).

Operating systems are important examples of concurrent systems. Air traffic control systems, mission critical systems, and real-time process control systems (such as those that control gasoline refineries, chemical manufacturing plants, and food processing plants) are other examples of processes that require parallel processing to achieve sufficient speed of execution.

Parallel processing does, however, bring about unique difficulties. Program logic is harder to follow for people trained to think in a linear fashion. It is difficult and time consuming to determine what activities can and cannot be performed in parallel. Parallel programs are much more difficult to debug than sequential programs. After a bug is supposedly fixed, it may be impossible to reconstruct the sequence of events that exposed the bug in the first place, so it would be inappropriate to certify, in some sense, that the bug has actually been corrected [6]. Since exhaustive debugging may not be possible, proving program correctness may become the standard for developing highly reliable, large-scale software systems [7] [8].

A.1 Flynn's Taxonomy

The past decade has witnessed a flood of parallel architectures. To assist in the classification of these computers, Flynn categorized machines by the number of instruction streams and data streams utilized [9]. The number of instruction streams includes only unique streams. For example, two processors executing the same instructions would be counted as a single instruction stream computer. However, the number of data streams includes all data being manipulated or simply the number of processors working (because each processor will be working on its own data stream). While Sillicorn [10] has extended this taxonomy from four to 28 different classes, only the four major divisions given by Flynn will be explained.

A.1.1 SISD

Single Instruction stream–Single Data stream (SISD) processors are the most commonly used today. These are uniprocessor computers with one memory that process one instruction at a time. Figure 2 illustrates the SISD architecture, with P denoting the processor and M representing memory, whether that be physically separate or contiguous. The interconnection network connects the processor and memory in any fashion.

This is essentially the von Neumann architecture, based on the original ideas of John von Neumann. For the basis of comparison, consider the following loop:

```
for (i=1; i<=n; ++i)
   c(i) = a(i) + b(i);
```

which might be converted into the following single instruction and data streams: Each line following down vertically indicates one time step, with the instruction stream following under the header of "Processor", and the data stream occurring as the arguments to each instruction.

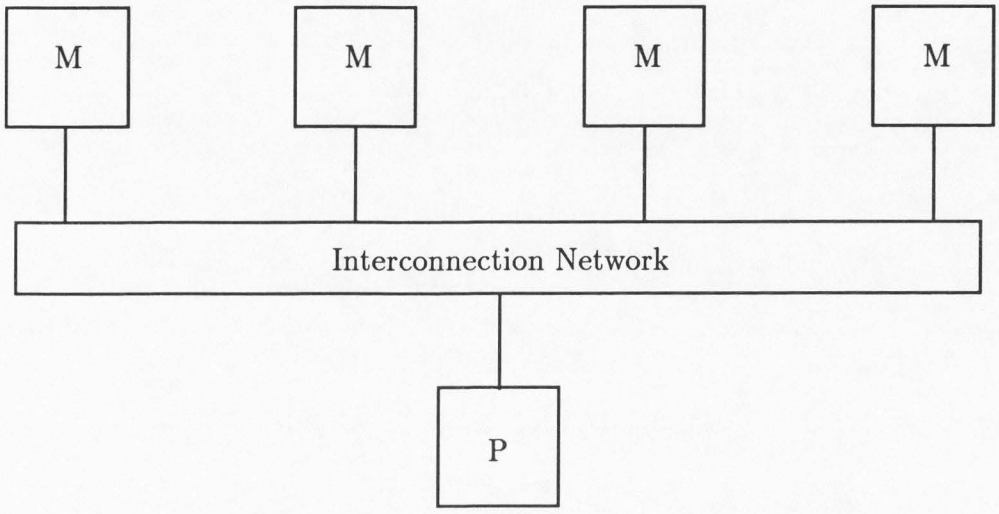

Figure 2: Uniprocessor Architecture

```
i = 1   LDA  R1, a(1)
        LDA  R2, b(1)
        ADD  R3, R2, R1
        STO  R3, c(1)
i = 2   LDA  R1, a(2)
        LDA  R2, b(2)
        ADD  R3, R2, R1
        STO  R3, c(2)
        |    |   |
        |    |   |
i = n   LDA  R1, a(n)
        LDA  R2, b(n)
        ADD  R3, R2, R1
        STO  R3, c(n)
```

As time proceeds, only one instruction can be carried out during each instruction cycle. Instructions are carried out upon the first element of the arrays, and then the process is repeated for the next corresponding element until all are processed.

A.1.2 SIMD

Single Instruction stream–Multiple Data stream (SIMD) machines employ multiple processors and, thus, multiple data streams; but each is performing the same

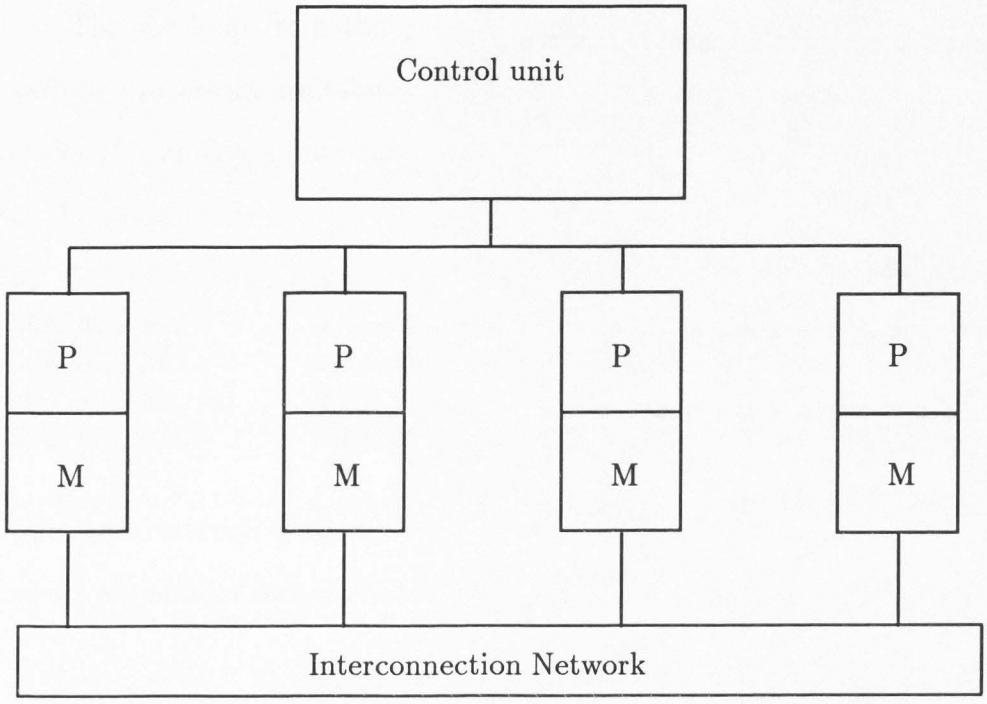

Figure 3: Array Processor Architecture

operations or using the same instruction stream. Both array processors and vector processors are commonly considered SIMD machines.

Array Processors An array processor contains multiple processing elements (PE's), which can be thought of simply as a processor and memory. One control unit broadcasts instructions simultaneously to all PE's. (See Figure 3.) Each PE receives the same instructions and operates on the date in its memory. While the individual processors may be relatively slow, the speed comes in the sheer number of processors available to work. For example, the Thinking Machines CM-2 utilized 65,536 processors in parallel.

The *for* loop from the previous example will demonstrate this method. The various processors are labeled P1 though Pn, with the instructions again following under the processor number and instructions on the same horizontal line occurring at the same time.

```
P1                  P2                  Pn
LDA R1, a(1)        LDA R1, a(2)        LDA R1, a(n)
LDA R2, b(1)        LDA R2, b(2)        LDA R2, b(n)
ADD R3, R2, R1      ADD R3, R2, R1      ADD R3, R2, R1
STO R3, c(1)        STO R3, c(2)        STO R3, c(n)
```

The same instruction stream is being broadcast to every processor, and each processor works only on its own data.

Vector Processors The other SIMD architecture, vector processor, functions differently. A vector processor consists of a control unit, one memory, and one processor with specialized functional units. The proper functional units are "pipelined" into a stream of operations to be performed on the data stream. As each unit completes its operation, its output is fed directly into the next unit. The parallel computation comes as the pipeline fills with data and each instruction in the pipeline executes concurrently on a different data element. Figure 4 shows the logical components of this architecture.

Let us go back to the familiar loop example:

```
LDA  VR1, a(1:n)
LDA  VR2, b(1:n)
ADD  VR3, VR2, VR1
STO  VR3, c(1:n)
```

This contains the special instructions for dealing with vector registers. The elements of a will begin to fill the first vector register while the elements of b will fill the second. Elements at corresponding positions will be added and stored. If more than

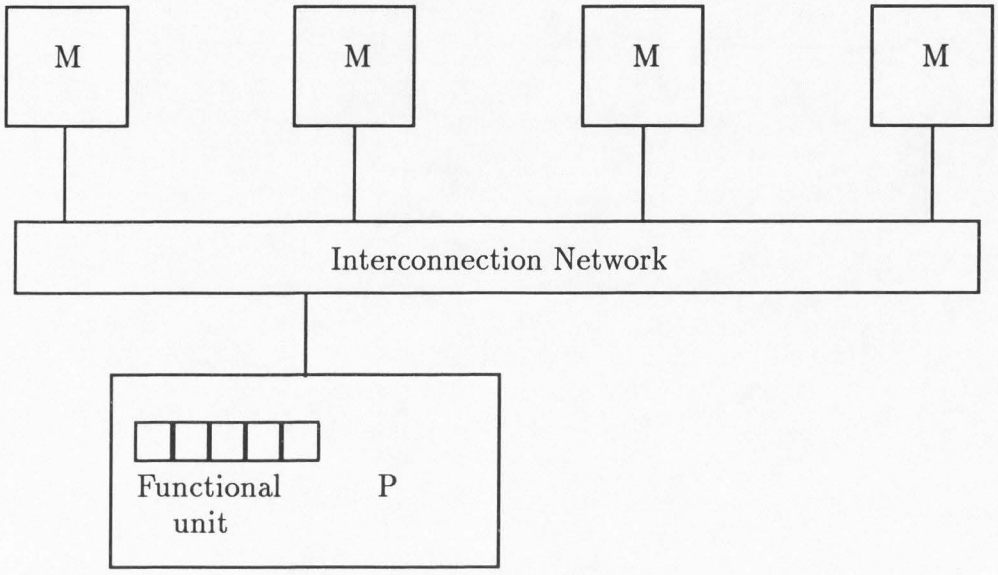

Figure 4: Vector Processor Architecture

four elements exist in the input vectors, then all four operations will occur in parallel, with each statement working one element behind its predecessor.

A.1.3 MISD

The Multiple Instruction stream–Single Data stream (MISD) has not found application in industry and is only mentioned for the sake of completeness.

A.1.4 MIMD

Multiple Instruction stream–Multiple Data stream (MIMD) processors are capable of truly independent parallel operation. Each processor may function fully and communicate with other processors in different ways. Just as SIMD architectures may be naturally subdivided, MIMD machines are most correctly classified as either shared-memory or local-memory machines.

Shared Memory Shared-memory machines consist of multiple processors connected to one common memory, and may be represented as in Figure 5. Any memory

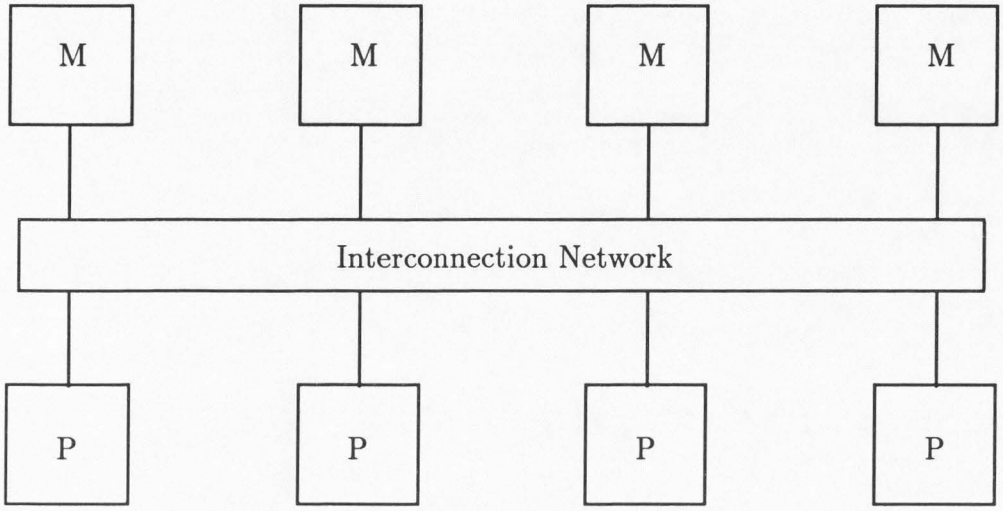

Figure 5: Shared-Memory Architecture

address is directly accessible for any processor. Because of this, programs may transparently access any address, without special concern as to the position of data in relation to the operating processor. Inter-process communication is very fast. The main pitfall for this method comes when shared data may be changed by any processor. If two processes try to modify the same data, each will write the new result. This and similar operations result in errors and require special precautions. To protect this shared modifiable data, schemes must be developed to ensure that each process needing the data is given exclusive access.

When considering shared-memory multiprocessors, many advantages become evident. Shared-memory models are perhaps the easiest to program, since any processor may access any memory location. The oldest parallel architectures are shared memory, so the programming environment is mature. Finally, although the memory is physically shared, it can mimic other architectures. Some disadvantages include the difficulty to scale bus-based machines beyond about 25 processors. Also, methods must be devised to ensure mutual exclusion of shared modifiable data.

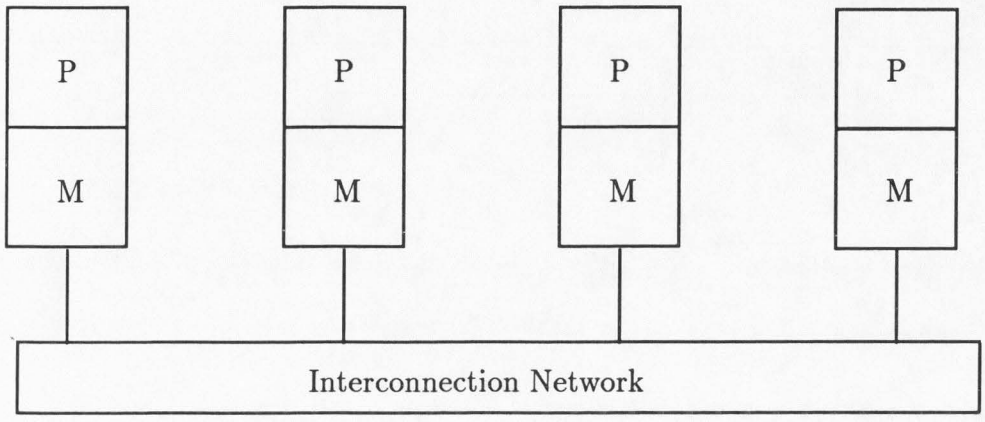

Figure 6: Local-Memory Architecture

The *for* loop would look similar for both models of MIMD machines. It is apparent that each processor is following a unique instruction and data stream.

```
P1                  P2                  Pn
LDA  R1, a(1)       LDA  R2, b(2)       STO  R3, c(n-1)
LDA  R2, b(1)       ADD  R3, R2, R1     LDA  R1, a(n)
ADD  R3, R2, R1     STO  R3, c(2)       LDA  R2, b(n)
STO  R3, c(1)       RTJ  LABEL          ADD  R3, R2, R1
```

Local Memory Unlike the shared-memory model, where a node is simply a processor, the node in a local-memory machine consists of a processor and memory. This is concisely represented in Figure 6. The memory is local to a particular processor, and outside references to it must be handled by passing a message to the owner of a particular reference. The owning processor will then access the data and send a message back containing the requested data. Because no data is shared, the problem of shared modifiable data is eliminated.

Workstations connected by a network and working in parallel fit this model. Local-memory machines are very scalable, and nodes may be added in an economic manner. The main disadvantages to using a local memory machine are the extra effort required to access data and the overhead, in the way of processing time, added by the messages.

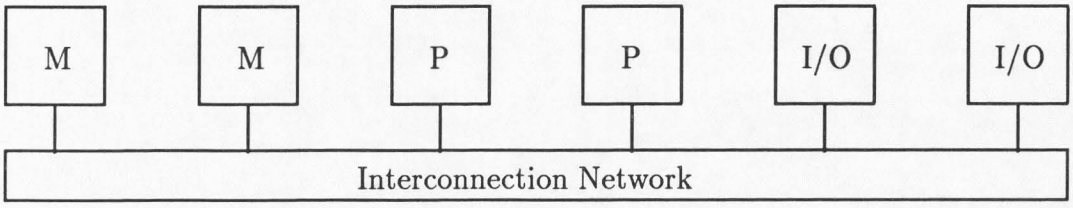

Figure 7: Shared Bus

Hybrid Architectures Machines have been developed to overcome the problems of both shared- and local-memory architectures. Many have shared but not equidistant memory. Not enough research has been done in this area, and it remains to be seen whether this approach will produce the best of both worlds or simply the worst of each.

A.2 Processor Interconnection Schemes

An inportant aspect in parallel architectures is that of the processor interconnect network. In shared-memory machines, every processor must have access to the common memory. To request information in a local memory architecture, every node must have a route of communication to every other node. For both types, each processing unit must also be connected to input/output processors, control units, and devices and must be controlled by one operating system.

A.2.1 Shared Bus

The shared-bus multiprocessor organization uses a single shared communication path between all processors, storage units, and I/O units (see Figure 7). The bus is a passive unit, with transfer operations between functional units controlled by bus interfaces on the units themselves. Ethernet local area networking uses this simple scheme.

The processor or I/O processor wishing to transfer data must check the availability of the bus and the availability of the destination unit and must initiate the

actual data transfer. The receiving units must be able to recognize which messages on the bus are addressed to them, and they must interpret and acknowledge the control signals received from the sending unit. Adding a new node is accomplished by simply connecting it to the bus. Then all that is needed is a software announcement to the other nodes that a new node is on the bus.

The primary disadvantage of this organizational scheme results from the single communications path. The bus can handle only one transmission at a time. The entire system will fail (catastrophic failure) if the bus fails. The transmission rate of the system is constrained by the transmission rate of the bus. Contention for the use of the bus in a busy system will degrade performance.

A.2.2 Crossbar-Switch Matrix

To circumvent the shared bus problems, a separate path can be added for every processor by increasing the number of buses, as shown in Figure 8. This method, the crossbar-switch matrix, can support simultaneous references to every storage unit without blockage. Certainly the crossbar-switch is the best-performing interconnection scheme.

The hardware to build such a switch, however, becomes increasingly complex as nodes are added. The switch must be capable of resolving conflicts for the same storage unit. Cost of the switch will increase as the product of the number of functional units and the number of storage units, effectively making this method inadequate for massively parallel systems.

A.2.3 Multistage Networks

In this compromise of the crossbar-switch matrix, multistage networks achieve good performance at a lower complexity. In this scheme, processors are connected to switches or "hubs", which are then connected to other hubs or switches. This is

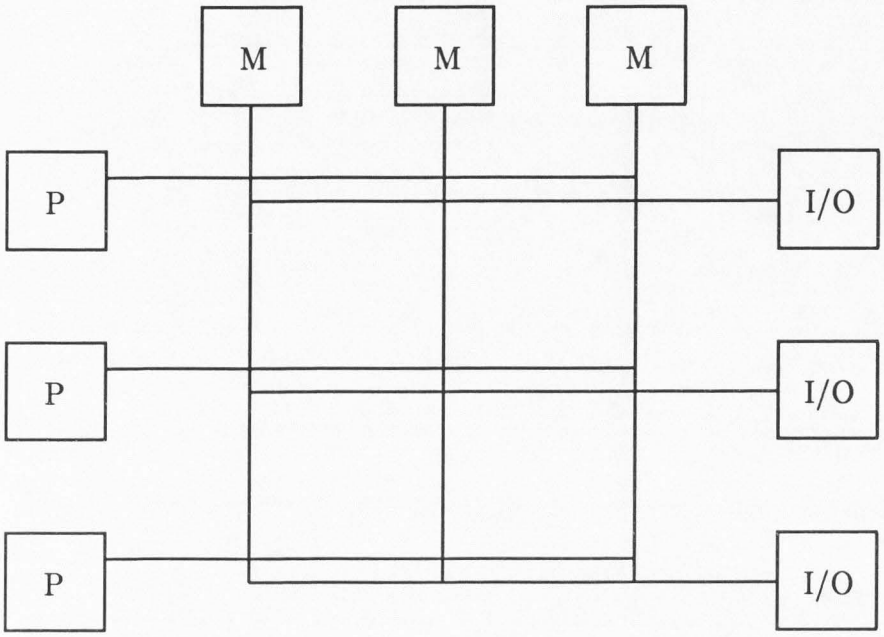

Figure 8: Crossbar

similar to the system used in the airline industry, which does not support flights from every destination to every other. Most flights from less-used locations are routed to a hub and from there to another hub or the final destination. Each unit is able to connect with any other unit, and the complexity of the interconnection scheme is greatly reduced for large numbers of processors. It is termed multistage because each message must pass through multiple switching elements to reach its destination.

A.2.4 Hypercube

The hypercube interconnection network connects large numbers of processors in a relatively economical manner. A two-dimensional hypercube is simply a square with a node at each corner. A three-dimensional hypercube is formed by connecting the corresponding points of two two-dimensional hypercubes, in this case, a cube with nodes on the corners. Higher-dimensional hypercubes are formed by connecting corresponding points on two hypercubes of the next lower dimension. Figure 9 sketches the several smaller dimensional hypercubes. Nodes reside at the endpoints of each solid

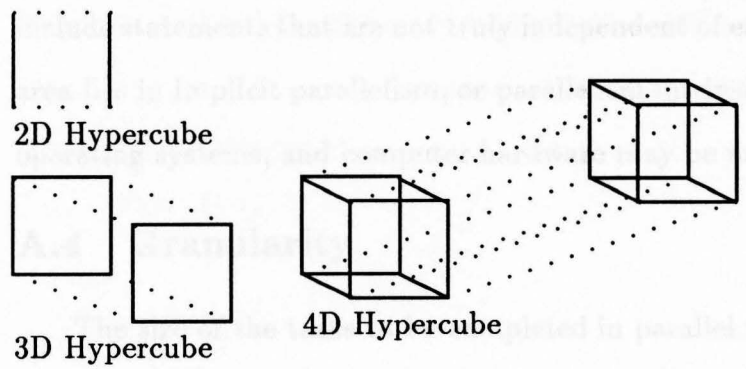

Figure 9: Hypercubes

line, with dotted lines connecting corresponding nodes. The savings of connections of this method can be easily seen in an example. For 8 nodes, a three-dimensional cube is used. A node can communicate with any other node via, at most, 3 connections. At the highest end of practical use, the CM-2 uses a 16-dimensional hypercube that supports 65,536 processors, with each being, at most, 16 connections from another.

A.3 Detecting Parallelism

Parallel detection is one important research possibility. This can be done by the programmer, language translator, hardware, or the operating system [11]. The shared memory model may use explicit or implicit parallelism. Explicit parallelism is indicated by a programmer using a concurrency construct such as cobegin/coend, as follows:

```
cobegin
    statement-1;
    statement-2;
    :
    statement-n;
coend
```

In a multiprocessing system, one processor would be assigned to each statement. Explicit parallelism is time consuming for the programmer, and the programmer may

include statements that are not truly independent of each other. The real hope in this area lies in implicit parallelism, or parallelism intrinsic to the algorithm. Compilers, operating systems, and computer hardware may be used to exploit this parallelism.

A.4 Granularity

The size of the tasks to be completed in parallel is referred to as its granularity. Fine-grained tasks include parallelizing individual assignment statements. Coarse-grained tasks consist of entire subroutines being executed in parallel.

A.4.1 Speedup and Efficiency

To determine the effect of adding processors to an effort, measure of the speedup and efficiency prove valuable. Speedup is defined as the time required to complete a program, using one processor divided by the time required using n processors. Perfect speedup is obtained when a program runs n times as fast when using n processors. Efficiency is defined as the speedup divided by the number of processors. Perfect speedup will result in 100 percent efficiency. Below is an example of 5 processors executing a program in 25 milliseconds; the program required 100 milliseconds when executed on one processor. The speedup and efficiency can be determined as shown.

$$S_5 = \frac{100}{25} = 4$$

$$E_5 = \frac{4}{5} = 80\%$$

A.4.2 Fine Grain

Directing fine-grained parallelism would be tedious and difficult. Luckily, the compiler is responsible for it. To achieve fine-grained parallelism, the compiler breaks individual statements into independent pieces and runs them concurrently.

Loop Distribution Often the statements within a loop are independent of each other. In such a case, one processor could be assigned to each iteration of the loop. The following statement

```
for (i=0; i<3; ++i)
    a[i] = b[i] + c[i];
```

would become

```
cobegin
    a[1] = b[1] + c[1];
    a[2] = b[2] + c[2];
    a[3] = b[3] + c[3];
coend
```

With n processors available, the time to execute such a loop with up to n elements is simply the time needed to calculate one element. In this way, data parallelism can be taken advantage of by true hardware parallelism.

Tree-Height Reduction Using rules of precedence, a processor may reorder an equation and indicate processes that may be executed concurrently in the object code. Often, a unique and sequential ordering is not needed. Using the rules of commutativity, associativity, and distributivity, compilers may rearrange expressions so that they are more amenable to parallel computation. Using the associative property of addition, the expression $((p+q)+r)+s)$ may be changed into $(p+q)+(r+s)$, which can be solved in fewer time steps because the first two additions may be carried out independently. The first equation requires three instruction cycles (from the three tree levels) while the second requires only two.

Using commutativity, $p+(q*r)+s$ goes to $(p+s)+(q*r)$, which again reduces from three levels of execution or dependence to two.

Finally, using distributivity, $p*(q*r*s+t)$ becomes $(p*q)*(r*s)+(p*t)$, which actually has added one more operation (five versus four) but reduced the tree height from four to three.

Never Wait A similar way of speeding mathematical computations involves the "never wait" rule. If a computation can be performed, it is better to do it now, even if it might not be valid later. If it is needed, computation is speeded.

```
a = c * c;
if (a==9)
    d = 10;
e = d * f;
```

The third statement does not rely at all on the first and may be performed in parallel with it. Should the second statement alter the value of `d`, then the third statement will need to be reevaluated. If `d` is not changed, then the third statement has already been performed; and the computation will complete faster.

A.4.3 Coarse Grain

Whole subroutines running independently and concurrently exhibit coarse- or large-grained parallelism. This is the type of parallelism employed by all MIMD machines. Because of the time wasted for communication, the larger the task to be performed in parallel, the faster the program will complete. If too few operations are directed by a synchronizing communication, the program may actually slow down.

Lusk [4] demonstrated the relative speedup and efficiency of a simple program which added two vectors, working element by element. Each addition was given to a requesting process, with the time required to complete the additions compared as the number of processors available was varied. The following gives the results.

Processes	microseconds
1	2980
2	5584
4	7201
8	10061
16	16159

For this problem, the execution time actually *increased* as processors were added. This is a result of the extremely small granularity of the program (one operation per processor request). Now, with the same program modified to artificially accomplish more work (the addition was repeated 10000 times), the results became

Processes	microseconds	speedup
1	5151046	1.00
2	2682454	1.92
4	1440593	3.58
8	826448	6.23
16	437237	11.78

When the granularity was increased (the load to each processor increased), the positive speedup was obtained.

A.5 Programming Models

Partly as a response to the different architectures, several different programming models have emerged. They differ in the level of parallelism exploited and the use of memory.

A.5.1 Loop-Level Parallelism

Automatic parallelizing compilers provide loop-level parallelism. They are used mainly in architectures with shared memory. While the programmer may still give compiler directives, the compiler is very good at parallelizing things such as the execution of a *DO* loop. The biggest problem for loop level parallelism comes from data dependence. This is demonstrated in the following example:

```
DO I=2,N
   A(I) = A(I-1) + B(I)
ENDDO
```

Each iteration relies on the previous computation. Data dependence defines a relation between two statements that imposes an order in which they must be executed. This occurs when the same memory location is used in both statements.

A notable example of a machine using this model is the Alliant FX/2800. It was specifically designed to determine, at compile time, if iterations of a loop were independent and then to exploit that situation by distributing the work of the loop across several processors. The result was a machine that could run old programs not written for any parallel operation and could spread the work out to achieve faster solutions. While speedups may be greater in programs written for parallel processors, this is a relatively easy way to attain parallel computation.

A.5.2 Shared Memory Model

Multiprocessors with a shared memory may use almost every programming model discussed. However, the most natural model for them is the shared-memory model. Large-grained parallelism is used as processors communicate through special variables. As long as no processor tries to change any of these special variables, no problems occur. However, when even one processor tries to change one of these special communication variables (known as shared modifiable data), indeterminate situations can arise. Assume the shared modifible variable `count = 7`. In parallel two processes execute:

```
Processor One                    Processor Two
count = count + 1                count = count - 1
```

for which the assembly code might look

```
Processor One                    Processor Two
LDA  R1, count                   LDA  R1, count
ADD  R1, 1                       SUB  R1, 1
STO  R1, count                   STO  R1, count
```

After this sequence of calls, would the variable count contain 6, 7, or 8?

The way to safely execute the previous example is to halt one processor while the other continues. Using a special lock variable, `LO = 0`, each can safely proceed.

```
Processor One                    Processor Two
CALL LOCK(LO)                    CALL LOCK(LO)
   count = count + 1                count = count + 1
CALL UNLOCK(LO)                  CALL UNLOCK(LO)
```

Only the first statement to call LOCK will be allowed to access the variable count. The other will be involved in a *busy wait*, meaning that it will perform no useful work while waiting for exclusive access to the variable count. Fortran style definitions would look like:

```
      SUBROUTINE LOCK(L)                SUBROUTINE UNLOCK(L)

100   CONTINUE                          L = 0
      IF (L .EQ. 1) THEN                RETURN
         GOTO 100                       END
      ELSE
         L = 1
         RETURN
      ENDIF
      RETURN
      END
```

When two or more processes attempt to access the variable count, they must first pass through the lock. The variable LO may be changed only by the LOCK and UNLOCK subroutines. The first process to access LOCK will set LO = 1. While process one works in its critical section (the section of code containing count, the shared modifiable data), any other process to enter the lock will continue to loop until process one calls subroutine UNLOCK and releases LO.

A.5.3 Message Passing Model

Each processor will have exclusive access to a portion of memory with the message passing model. A processor may reference data in its own memory, but all other references must be handled by message passing. A node needing a variable from

another processor would first request the location from the owning processor. That processor would retrieve the information and then send it back to the requesting node.

This method eliminates the need of protecting shared variables but introduces other problems. Most significant of these is the overhead, in the way of processing time, required to send messages. The overhead required, however, is proportional to the grain of parallelism (or the size of tasks going on in parallel). With fine-grained parallelism, parallelism on a small level (such as parts of a single assignment occurring concurrently), the cost of message passing greatly overshadows any gain obtained by parallelism. The time required to send the messages to coordinate the effort may become greater than the time required for one processor to complete the entire task. However, on large-grained problems (whole subroutines working concurrently), the time to pass messages becomes inconsequential. In other words, when the time required to send the message is small compared to the task specified to be performed by that message, then the message itself is not a performance issue. A good example of this is a message indicating a whole subroutine to run.

The reasons to use message passing are numerous. First, is the applicability of the code. Message-passing programs were designed to accommodate local memory architectures, but they can also be run on shared-memory architectures. For shared-memory machines, the shared memory is divided between the processors; and communication between processors then takes place only through messages. While shared-memory machines may run message-passing programs, local-memory machines may not easily run shared-memory programs. Thus, message-passing programs may run on the widest range of parallel architectures.

Second, these programs run efficiently on both shared- and local-memory architectures. Message passing encourages programs that have a very large grain. Since each message carries a penalty of wasted time, the number of messages per block of

directed work decreases. Also, message passing takes advantage of data locality. The data on which each processor will work is predominately in its own memory space.

Finally, the template of a message-passing program is very easy to understand. Every node will run the same program, with one node acting as a master and all the remaining nodes acting as slaves. The master is responsible for user input and output and orchestrating the work done by slave processes. Consider the following:

```
#include "includes.h"
main()
{
   init();
   if (get_my_id() == 0)
      master();
   else
      slave();
   cleanup();
}
```

The function get_my_id() returns a unique positive integer for each processor. The first process to call it will become the master and enter the master() function, where it will usually determine the total number of processors, divide the work, and broadcast the needed information to all other nodes. All other processors will enter the slave() function and wait for instructions from the master. Any message-passing program is simply an expansion of the above.

A.5.4 Cluster Model

This software model is a combination of both the shared-memory and message-passing models. Loosely coupled clusters of processors work together. Each processor within a cluster communicates through shared variables in a common memory, while communication among clusters is achieved by message passing. This model has not yet seen general acceptance, but seems to be the most plausible for the future. As

it becomes possible to include more than one processor on a physical chip, this programming model will be most naturally implied by the hardware.

References

[1] Lusk, E., *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[2] Kuck, D. J., and A. H. Sameh, *A Supercomputing Performance Evaluation Plan,* Tech. Rep. 692, Center for Supercomputing Research and Development, University of Illinois, June 1987.

[3] Dongarra, J. J. and J. R. Bunch, C. B. Moler, G. W. Stewart, *LINPACK Users' Guide.* Philadelphia: the Society for Industrial and Applied Mathematics, 1979.

[4] Berry, M., et al., *The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers,* The International Journal of Supercomputer Applications, 3 (1989), pp. 5–40.

[5] Mosher, C. C., S. Hassanzadeh, and F. G. Lou, *ARCO Seismic Benchmark Users Guide*, Version 1.00, January 1992.

[6] Myers, W., "Can Software for the Strategic Defense Initiative Ever Be Error-Free?" *Computer*, Vol. 19, No. 11, November 1986, pp. 61–68.

[7] Keller, R. M., "Formal Verification of Parallel Programs," *Communications of the ACM,* Vol. 19, No. 7, July 1976, pp. 371–384.

[8] Lamport, L., "Proving the Correctness of Multiprocess Programs," *IEEE Transactions on Software Engineering,* Vol. SE-3, No. 3, March 1977, pp. 125–143.

[9] Flynn, M. J., "Very High–Speed Computing Systems," *Proceedings of the IEEE*, Vol. 54, December 1966, pp. 1901-1909.

[10] Sillicorn, D. B., "A Taxonomy for Computer Architectures," *Computer*, Vol. 21, No. 11, November 1988, pp. 46-57.

[11] Kuck, D. J., "A Survey of Parallel Machine Organization and Programming," *ACM Computing Surveys*, Vol. 9, No. 1, March 1977, pp. 29-60.

[12] Margulis, N., *i860 Microprocessor Architecture.* Osborne McGraw-Hill, 1990.

[13] Perrott, R. H., *Parallel Programming.* Addison-Wesley Publishing Company, 1987.

[14] Stallman, R., *GNU Emacs Manual.* Free Software Foundation, 1986.

[15] Lamport, L., LaTeX: *A Document Preparation System.* Addison-Wesley Publishing Company, 1986.

[16] Deitel, H. M., *An Introduction to Operating Systems.* Addison-Wesley Publishing Company, 1990.

[17] Carey, G. F., *Parallel Supercomputing: Methods, Algorithms, and Applications.* John Wiley and Sons, 1989.

[18] Cosnard, M., *Parallel Algorithms and Architectures.* Elsevier Science Publishing Company, Inc., 1986.

[19] Hack, J. J., "Peak vs. Sustained Performance in Highly Concurrent Vector Machines," *Computer*, Vol. 19, No. 9, September 1986, pp. 11-20.

[20] Bertsekas, D. P. and J. N. Tsitsiklis., *Parallel and Distributed Computation.* Englewood Cliffs, NJ: Prentice Hall, 1989.

[21] Bhuyan, L. N., Q. Yang and D. P. Agrawal, "Performance of Multiprocessor Interconnection Networks," *Computer*, Vol. 22, No. 2, February 1989, pp. 25-37.

[22] Burns, D., "Loop-Based Concurrency Identified As Best At Exploiting Parallelism," *Computer Technology Review*, Vol. 8, No. 16, Winter 1988, pp. 19-23.

[23] Carlson, W. W. and K. Hwang, "Algorithmic Performance of Dataflow Multiprocessors," *Computer*, Vol. 18, No. 12, December 1985, pp. 30-43.

[24] Fox, G., "Use of the Caltech hypercube," *IEEE Software*, Vol. 2, July 1985, p. 73.

[25] Hillis, D., *The Connection Machine* Cambridge, Ma: MIT Press, 1985.

[26] Kuck, D. J., "A Survey of Parallel Machine Organization and Programming," *ACM Computing Surveys*, Vol. 9, No. 1, March 1977, pp. 29-60.

[27] Chandy, K. M. and S. Taylor, *An Introduction to Parallel Programming.* USA: Jones and Bartlett Publishers, Inc., 1992.

[28] Foster, I. and S. Taylor, *Strand: New Concepts in Parallel Programming.* Englewood Cliffs, NJ: Prentice Hall, 1990.

P4 Versions of the `fdmod` and `seis` Perfect-Seismic Codes

David Levine and D. Ladd Williamson

B Users Guide

B.1 Introduction

Guided by a proposal to create a benchmarking suite given by Kuck and Sameh[1], academic and industrial collaborators initiated the Perfect (**Per**formance **E**valuation for Cost-Effective **T**ransformations) benchmarking group in 1987. The goal was to produce a large number of applications which could be used in performance evaluation and ported to a large number of vector parallel high performance computing machines[2].

This effort was largely successful, with 13 programs representing a variety of scientific and engineering problems. These codes were successfully ported to over 30 machines. This effort proved to be the first systematic solution to the problem of benchmarking parallel machines.

The Perfect-2(P-2) codes are being developed to provide benchmarks of Massively Parallel Processing (MPP) computers. Eventually, three versions of every P-2 code will be included in the distribution: Fortran 77 for loop-level parallelism, Fortran 90, and Fortran 77 + message passing. P-2 is eventually intended to encompass approximately 10-15 applications. Among these are the ARCO benchmarks[3], it is these which we are focusing on here. We are assisting in this project by providing p4 message passing versions that can run on most MPP machines and workstation networks.

B.2 P4

P4 is a collection of portable functions written in C to provide a basis for message passing[4]. A Fortran interface provides applicability for that language. The complete distribution of the latest version of p4 may be freely obtained by anonymous ftp from `info.msc.anl.gov` in the directory `pub/p4`.

Three files must be included in the directories of each program. First, `p4f.h` declares certain constants and types p4 function calls. Second, `procgroup` specifies the number and location of each process to be started. See the machine specific notes section. Finally, the `Makefile` must be modified to point to the p4 directory to link in specific libraries. Examples showing different flags required will be shown in the machine specific notes section below.

Several versions of p4 exist, with the most recent being p4-1.2b. This contains many improvements over p4-1.2, the most important to Perfect-Seismic being the exclusion of a special `slave()` function. Version 1.2 required such a subroutine, and slave processes could not exist outside of it. The newer p4-1.2b allows for more natural programs by not imposing the `slave()` subroutine.

Within p4 is the ability to log user-specified events using `alog` subroutines. The resulting files, called logfiles, may be examined using `Upshot`, available from the same directory as p4 by anonymous ftp[5].

B.3 Directory Structure

Information reguarding the Directory Structure may be obtained from the Arco Seismic Benchmark Users Guide[3]. An environment variable "`BENCH`" must be set to the top-level directory. Two other environment variables must be set to compile, link, and run correctly. The first, "`ARCH`", is the architecture of the machine where compilation will occur (we used "sun4", "fx2800", "ksr", "symmetry", "ncube", "tc2000", "rs6000", and "cm5" for the Sun SPARC, Alliant FX/2800, Kendall Square KSR-1, Sequent Symmetry, nCUBE 6400, BBN TC-2000, IBM RS6000, and Thinking Machines' CM-5, respectively). The second, "`TARGET_ARCH`" is the architecture of the machine being compiled for. These variables determine where to build and search for various libraries and which makedef file is to be included for machine specific rules. One makedef file exists for each different architecture, and contains all the necessary flags for compiling on different machines. We used several architectures ("sun4", "gamma", "delta", "fx2800", "ksr", "symmetry", "ncube", "tc2000", "rs6000", and "cm5") but more could be used by adding the corresponding subdirectories and creating a specific makedef file. To support a new architecture, the user would need to create a makedef file for the desired machine which included all machine specific switches. The `bin` subdirectories consist of all possible architectures on which to run ("TARGET_ARCH"). The `lib` directory functions similarly. The user must create the subdirectories under the `bin` and `lib` directories.

The `doc` directory contains the Perfect-Seismic Users Manual, and the makedef files are in the `include` directory. The source code is found under `src`, with `src/f77` containing the sequential versions of the applications, and `src/msg` containing the p4 versions.

As an example, consider a user installing the software in

```
/home/williams/Benchmarks/PERFECT
```

to be run on an Intel DELTA and cross-compiled on a sun4. The environment variables should be set as follows:

```
% setenv BENCH /home/williams/Benchmarks/PERFECT
% setenv ARCH sun4
% setenv TARGET_ARCH delta
```

Also, the following directories must be created:

```
% mkdir $(BENCH)/bin/delta $(BENCH)/lib/delta
```

B.4 The Applications

B.4.1 Fdmod

`fdmod` performs 3D finite difference modeling with the acoustic wave equation. The directory containing this is `$(BENCH)/src/msg/fdmod`.

Files `fdmod` is made up of seven Fortran source files, `p4` header and procgroup files, and an input file.

1. **fdmod.f** is the driver program for `fdmod`. It initializes the environment, creates the slave processes, and begins main processing routines.
2. **doit.f** initializes various arrays, and then enters a loop to do all the calculations. Upon completion of the big loop, this subroutine prints out the performance results.
3. **getparm.f** instructs the master node to read the parmfile and broadcast the values to all slave nodes.
4. **bcs.f** contains the code for absorbing boundary conditions.
5. **fdoper.f** is the actual finite difference operator.
6. **source.f** is the Ricker's source function code.
7. **vclr.f** zero's a vector. This is copied from the Standard Seismic Subroutine Library.
8. **parmfile** contains the size of the model and various other input parameters.
9. **procgroup** specifies to `p4` which machine(s) to use.
10. **p4f.h** types the `p4` functions.

To run `fdmod`, care must be taken to assure that the size of the model is small enough to fit in the memory of a single processor. This is because currently whole arrays are allocated even though only part is used by each processor. The number of depths per level is given by the first three parameters of the `parmfile`. For one node on all tested machines, a safe size is 50 per dimension, or 50 for each of the first three parameters in the `parmfile`.

To determine the maximum model size allowed, divide the available memory size in bytes by the size of a `real` variable in bytes times three (`fdmod` uses three large

arrays of type `real`). If M is the available memory size and R is the size of the type `real` then the model size is found by $\frac{M}{3 \times R}$. This is the maximum size. To determine the maximum cubic model, take the cube root of the answer. These numbers will be the first three arguments in the parmfile.

Changes and Bug Fixes The parallel version of `fdmod` was written by Siamak Hassanzadeh. Ted Charrette based his parallel version of `fdmod` on Hassanzadeh's code. Charrette's code is the source of our work.
Feb 9, 1993: Alog calls removed from the code. The newer calls were added, and then removed because the final state of the calls is still in flux. The non-standard p4 calls have been replaced with similarly functioning standard p4 calls.
May 3, 1993: Makefiles have been standardized, and now with very few exceptions, only changes to the path of p4 in the makedef files are required to compile and link on any supported architecture.

Future Work Currently, the entire arrays are allocated on each processor, even though only part is used. This needs to be rewritten to only allocate the memory required per processor. Also, `fdmod` is only decomposed in one dimension, Alberto Roveda is working on a three-dimsional decomposition.

Input and Output Files The input file `parmfile` contains the dimensions of the model and other parameters. The order of input variables is `nx, ny, nz, nxs, nys, nzs, nstep, ioper, sf, dx, dt.`

The output comes to the screen. A file has been left in the code which may be used for the output instead, by changing the unit number on the write statements in `doit.f` from "*" to 9, corresponding with the file `fdmod.out`.

B.4.2 Seis

`seis` performs prestack seismic processing. An originating process reads seismic traces, which are then piped through a chain of data processing routines. A final process writes the processed traces out to disk or tape.

Files This version of `seis` includes the following processes:

DCON – seismic trace deconvolution
DGEN – synthetic data generation
DMOC – dip moveout correction
FANF – 2D spatial filtering by Fourier transform
GEOM – seismic geometry specification
NMOC – normal moveout correction
READ – read seismic benchmark file

RTMG – reverse time finite difference migration
STAK – stack seismic traces
WRIT – write seismic benchmark file

These processes are in `$(BENCH)/src/msg/seis`.

Input and Output Files Two small sample problems are supplied in the parameter files `small1.prm` or `small2.prm` in the `seis` directory. These files contain a sequence of processes to be run that produce seismic trace files. The path will need to be modified on the line following the `WRIT` line in `small1.prm` and the lines following both `READ` and `WRIT` in `small2.prm`. Then modify the path in the script `run.small` to point to the same place. The following script will then run both small tests, of which the second is dependent on the resulting seismic trace files of the first.

```
rm -f /home/williams/tmp/stest?.*
cp small1.prm seis.prm
seis -p4pg procgroup_file_name
cp small2.prm seis.prm
seis -p4pg procgroup_file_name
```

The first line deletes any previous seismic trace files. Any old files would cause a file creation error. The first parameter file is copied into `seis.prm`, which is looked for by `seis`. Then the program is started with a specific procgroup file.

The `utils` directory contains many subroutines for managing seismic files. A `make` command in this directory will cause an archive called `libbench.a` to be created. A `make` in the `seis` directory will create an archive called `libseis.s`.

Changes and Bug Fixes March 12, 1993: The YAMPL message passing calls have been replaced by `p4` calls. Most C functions used by YAMPL have been removed or replaced by Fortran versions.
May 3, 1993: Makefiles have been standardized, and now only changes to the path of p4 in the makedef files are required to compile and link on any supported archtecture.

Future Work Remove the final C function `jloc` which returns the addresses of a common block.

B.4.3 General Make Notes

All Makefiles are machine-independent. Changes are made in the `$(BENCH)/include/makedef.$(TARGET_ARCH)` files. To compile either `fdmod` or `seis`, first edit the makedef.$(TARGET_ARCH) and set the `MACHINE` and `P4_HOME_DIR` variables. `fdmod` can then be compiled using the usual UNIX make in its directory. The

`utils` directory must be compiled *before* `seis`, as many routines call the special seismic I/O routines, etc. Perform a UNIX make first in the `utils` directory, and then in `seis`.

A special notation may be used to keep multiple copies of the procgroup file specifing different numbers of processes. These are written `$(ARCH)#.pg`. E.g., the KSR-1 with procgroups specifying one, two, and sixteen processes would have files `ksr1.pg`, `ksr2.pg`, and `ksr16.pg`.

Run scripts are included in each `$(BENCH)/bin/$(TARGET_ARCH)` directory. Usually the only modification needed for each run script is to change the name of the procgroup.

B.5 Local Memory Machines

For most local memory machines the procgroup will require only one line which contains the string "`local`", the number of slave processes, and the path to the executable. The fast communication of the machine between nodes is then used.

B.5.1 Intel DELTA

The Intel DELTA is a local memory machine with 528 nodes (each consisting of an i860 processor and 16MB of memory) connected in a 2–D mesh topology. Each node is connected to a Mesh Routing Chip which controls message passing.

Compilation The machine-specific file `makedef.delta` which assumes compilation on a sun4 contains the following variables,

```
MACHINE = DELTA
P4_HOME_DIR = /usr/local/p4-1.3/$(MACHINE)

OTHER = -lkmath -node
XLIB = -lX11 -lsocknode
ARCH = sun4

CC = icc
FC = if77
CFLAGS = $(COPT)
FFLAGS = $(FOPT)
COPT =
FOPT = -O4 -Mvect -Mnodepchk

ARFLAGS = r
AR = ar860
AS = as
```

```
LIBS = $(P4_HOME_DIR)/lib_f/libp4_f.a $(P4_HOME_DIR)/lib/libp4.a
LIB = $(BENCH)/lib/$(TARGET_ARCH)/libbench.a
SLIB = $(BENCH)/lib/$(TARGET_ARCH)/libseis.a
BIN = $(BENCH)/bin/$(TARGET_ARCH)

P4_CFLAGS = -I$(P4_HOME_DIR)/include
P4_FFLAGS = -I$(P4_HOME_DIR)/lib_f

.c.o:
$(CC) $(CFLAGS) -c $<

.c.a:
$(CC) $(CFLAGS) -c $<
$(AR) $(ARFLAGS) $@ $*.o
 rm -f $*.o

.f.o:
$(FC) $(FFLAGS) -c $*.f

.f.a:
$(FC) $(FFLAGS) -c $*.f
$(AR) $(ARFLAGS) $@ $*.o
rm -f $*.o

.s.o:
$(COMPILE.s) -o $@ $<
.s.a:
$(COMPILE.s) -o $% $<
$(AR) $(ARFLAGS) $@ $%
$(RM) $%
COMPILE.s= $(AS) $(ASFLAGS)
```

Procgroup File The DELTA procgroup file (`delta1.pg`) to run `seis` on one node is:

```
local 0
```

To run on two nodes (delta2.pg) use:

```
local 1 /usr2/levine/seis
```

To run on 16 nodes (delta16.pg) use:

```
local 15 /usr2/levine/seis
```

Execution On the DELTA, the user must use the `mexec` command to allocate a partition and begin execution. Process 0 must be loaded on the first node with a parameter specifying the procgroup file to use. The remaining nodes are loaded with a switch to indicate that they are slave processes. For example, this command (see shellscript `run1`) loads one process on logical node "0".

```
mexec -t "(1,1)" -f "0 seis -pg delta1.pg"
```

This command (see shellscript `run16`) loads the "master" process on logical node "0", and the remaining processes on logical nodes 1-15.

```
mexec -t "(4,4)" -f "0 seis -pg delta16.pg" -f "1-15 seis -amp4slave"
```

Special Notes Be sure that the environment variables `PGI` and `IPSC_XDEV` are pointing to the correct directories.

The path in the procgroup file is very likely to be different than the structure compiled on. Be careful of pathname when moving from the front-end where compilation is done to the mesh for execution.

Current Status May 24, 1993: Both `fdmod` and `seis` compile, link, and run on the DELTA.

B.5.2 Intel iPSC/860

The Intel iPSC/860 is a local-memory machine connected in a hypercube topology. The particular machine used was an 8-node system at Argonne. Each node consists of an Intel i860 microprocessor and 16MB of memory. The front-end is an Intel 386 computer. Compilation was done on a Sun Sparc.

Compilation The machine-specific file `makedef.ipsc860` contains the following variables, of which only the first two lines will need to be modified before compilation.

```
MACHINE = IPSC860
P4_HOME_DIR = /usr/local/p4-1.3/$(MACHINE)

OTHER = -lkmath -node
XLIB = -lX11 -lsocknode
ARCH = sun4

IPSC_XDEV = /usr/local
BIN860 = $(IPSC_XDEV)/i860/bin.$(ARCH)
INC860 = $(IPSC_XDEV)/i860/include-ipsc
AR = $(BIN860)/ar860
```

```
ARFLAGS = r
CC = $(BIN860)/icc
IFLAGS = -I$(INC860)
CFLAGS = $(COPT) -i860 -Dipsc860 -DIPSC $(IFLAGS)
FFLAGS = $(FOPT) -i860
XFLAGS = -lX11 -lsocknode
COPT =
FOPT = -O4 -Mvect -Mnodepchk
FC = $(BIN860)/if77
AS = $(BIN860)/as860
FLINKER = $(FC)

LIBS = $(P4_HOME_DIR)/lib_f/libp4_f.a $(P4_HOME_DIR)/lib/libp4.a
LIB = $(BENCH)/lib/$(TARGET_ARCH)/libbench.a
SLIB = $(BENCH)/lib/$(TARGET_ARCH)/libseis.a
BIN = $(BENCH)/bin/$(TARGET_ARCH)

P4_CFLAGS = -I$(P4_HOME_DIR)/include
P4_FFLAGS = -I$(P4_HOME_DIR)/lib_f

.c.o:
$(CC) $(CFLAGS) -c $<

.c.a:
$(CC) $(CFLAGS) -c $<
$(AR) $(ARFLAGS) $@ $*.o
rm -f $*.o

.f.o:
$(FC) $(FFLAGS) -c $*.f

.f.a:
$(FC) $(FFLAGS) -c $*.f
$(AR) $(ARFLAGS) $@ $*.o
rm -f $*.o

.s.o:
$(COMPILE.s) -o $@ $<
.s.a:
$(COMPILE.s) -o $% $<
$(AR) $(ARFLAGS) $@ $%
$(RM) $%
```

```
COMPILE.s= $(AS) $(ASFLAGS)
```

Procgroup File For the iPSC/860, the procgroup file is the string "`local`, the number of slaves, and the full pathname to the executable. The following was used for one node on the iPSC/860 at ANL.

```
local 0
```

and the procgroup for eight nodes is:

```
local 7 /home/williams/PERFECT/bin/ipsc860/fdmod
```

Execution To run, first a cube must be allocated, the program loaded and run, and then the cube cleaned up and released. The script `run2` contains the following commands to run on 8 nodes using the procgroup file `p4gamma8.pg`.

```
getcube -t8
load 1-7 /home/williams/PERFECT/bin/ipsc860/fdmod -amp4slave
load 0 /home/williams/PERFECT/bin/ipsc860/fdmod -pg gamma8.pg;waitcube
killcube
relcube
```

B.5.3 Special Notes

Remember to always cross-compile on a Sun.

Any path beginning with `/cfs` uses the "concurrent file system". This is a fast I/O system accessible only through the nodes, not the front-end. No references to the `/cfs` currently exist in the benchmark suite.

Current Status May 24, 1993: Both `fdmod` and `seis` compile, link, and run.

B.5.4 nCUBE

The nCUBE is a local memory machine in a hypercube topology.

Compilation The machine-specific file `makedef.ncube` contains the following variables,

```
MACHINE = NCUBE
P4_HOME_DIR = /home/users/cust/williams/p4-1.3

AR = nar
ARFLAGS = r
CC = ncc
```

```
CFLAGS = $(COPT)
FFLAGS = $(FOPT)
COPT = -O3 -BX
FOPT = -O3 -BX -ncube2s
FC = nf77
RAN = ranlib
FLINKER = $(FC)

LIBS = $(P4_HOME_DIR)/lib_f/libp4_f.a $(P4_HOME_DIR)/lib/libp4.a
LIB = $(BENCH)/lib/$(TARGET_ARCH)/libbench.a
SLIB = $(BENCH)/lib/$(TARGET_ARCH)/libseis.a
BIN = $(BENCH)/bin/$(TARGET_ARCH)

P4_CFLAGS = -I$(P4_HOME_DIR)/include
P4_FFLAGS = -I$(P4_HOME_DIR)/lib_f

.c.o:
        $(CC) $(CFLAGS) -c $<

.c.a:
        $(CC) $(CFLAGS) -c $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o

.f.o:
        $(FC) $(FFLAGS) $(FOPT) -BX -O3 -c $*.f

.f.a:
        $(FC) $(FFLAGS) $(FOPT) -BX -O3 -c $*.f
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o

.s.o:
        $(COMPILE.s) -o $@ $<
.s.a:
        $(COMPILE.s) -o $% $<
        $(AR) $(ARFLAGS) $@ $%
        $(RM) $%
COMPILE.s= $(AS) $(ASFLAGS)
```

Procgroup File The procgroup file contains the string "local", the number of slave processes, and the full path to the executable. An example procgroup file for one node

running `fdmod` is:

```
local 0 /visitors/levine/PERFECT/bin/ncube/fdmod
```

For 16 nodes running `seis` use:

```
local 15 /visitors/levine/PERFECT/bin/ncube/seis
```

Note that even with only one node, the full pathname must be specified.

Execution The shell script `runcube` in the `$(BENCH)/bin/ipsc860` directory will execute any `p4` program. It takes as arguments the procgroup file name and any parameters to be passed to `p4`. From the procgroup file the number of processors to be used is determined, the path is found, and a cube is allocated, loaded, and execution is initiated. To maintain consistency, `runcube` is called from the usual `run.fdmod` and `run.seis` scripts.

Special Notes Many utilities are available, just prepend an "n" to whatever you are looking for. For example, the link editor is `nld`, the archiver is `nar`, and the debugger is `ndb`. To find on-line information, use `nman`.

One C function remains in `seis`. To compile, change the name of the function to all uppercase, and remove the trailing underscore.

Current Status May 24, 1993: `fdmod` will run with only one process, and complains about an overfull message buffer with two processes. This ran the first time around.
May 24, 1993: `seis` compiles. When run with small1.prm, it works fine, but when used with small2.prm, it crashes early (during file creation).

B.5.5 IBM SP-1

The SP-1 is a local memory machine with nodes consisting of IBM RS6000 processors and a 128MB memory. The configuration used in our testing used Ethernet connectivity.

Compilation For compilation, follow exactly the procedure for compiling for a network of RS6000's described in Section B.6.2. The SP-1 is binary compatible with the RS6000.

Procgroup File The procgroup file to run on one node is:

```
local 0
```

For four nodes, one line must be included for each process (at the time of this writing the fast switch was not installed and Ethernet was used for message passing).

```
local 0
spnode4 1 /u/williams/seis
spnode5 1 /u/williams/seis
spnode6 1 /u/williams/seis
```

Execution No prior allocation of nodes is necessary for the SP-1, nodes are specified in the procgroup file. To execute on four nodes using the procgroup in the previous Section use

```
rlogin spnode3
cd /u/williams
rm -f /u/williams/stest?.*
cp small1.prm seis.prm
seis -p4pg /u/williams/seis
cp small2.prm seis.prm
seis -p4pg /u/williams/seis
```

Special Notes In our test machine, the nodes were connected via Ethernet. However, a switch is coming which will speed message passing greatly.

Current Status May 24, 1993: The first port of `fdmod` was successful, but this time it crashes right at the start.
May 24, 1993: `seis` runs fine, but without the fast switch, slows as processors are added.

B.5.6 CM-5

The CM-5 is a local memory machine with nodes connected in a Fat-Tree topology. To most effectively use the CM-5, jobs must be submitted to the distributed job manager (DJM). Although smaller p4 programs have been run on the CM-5, we have been unable to successfully run either `fdmod` or `seis` there.

Compilation A complete makedef file has not been completed, but here are some of the appropriate definitions:

```
CC = cc
FC = cmf
CLINKER = cmmd-ld -comp cc
FLINKER = cmmd-ld -comp f77
```

Procgroup File For the CM-5, the procgroup consists of the string "local", the number of slave processes, and the full pathname to the executable. For one node, use:

```
local 0
```

and for 4 nodes use a procgroup of the form:

```
local 3 /u/ncsa/dschneid/PERFECT/bin/cm5/fdmod
```

Execution To submit jobs to the Distributed Job Manager, do the following:

```
% jsub myjob
Number of processors (8K)?
Estimated CPU time (5min)?
Estimated memory (128M)?
Job submitted successfully. Job id is 43.
```

Special Notes

Current Status May 20, 1993: Nothing important has compiled.

B.6 Networks of Workstations

A network of workstations can be viewed as a loosely coupled local memory machine with very poor communication latencies and bandwidths. Since p4 programs are easily ported, program development often occurs by running first on workstation networks, and then porting debugged p4 code to other machines.

B.6.1 SUN Network

Compilation The machine-specific file `makedef.sun4` contains the following variables,

```
#
#  ARCO Seismic Benchmarks
#  Additions to make rules for Sun SPARC
#  For sun4, add ranlib to make rules for libraries
#

MACHINE = SUN
P4_HOME_DIR = /usr/local/p4-1.3/$(MACHINE)

XLIB = -lX11
```

```
RAN = ranlib

CFLAGS = -D$(ARCH) $(COPT)
FFLAGS = $(FOPT)
FOPT = -O3
COPT = -O3

FLINKER = $(FC)

ARFLAGS = ruv
LIBS = $(P4_HOME_DIR)/lib_f/libp4_f.a $(P4_HOME_DIR)/lib/libp4.a
LIB = $(BENCH)/lib/$(TARGET_ARCH)/libbench.a
SLIB = $(BENCH)/lib/$(TARGET_ARCH)/libseis.a
BIN = $(BENCH)/bin/$(TARGET_ARCH)

P4_CFLAGS = -I$(P4_HOME_DIR)/include
P4_FFLAGS = -I$(P4_HOME_DIR)/lib_f

.c.o:
        $(CC) $(CFLAGS) -c $<


.c.a:
$(COMPILE.c) -o $% $<
$(AR) $(ARFLAGS) $@ $%
$(RAN) $@
$(RM) $%

.f.o:
        $(FC) $(FFLAGS) -c $<

.f.a:
$(COMPILE.f) -o $% $<
$(AR) $(ARFLAGS) $@ $%
$(RAN) $@
$(RM) $%
```

Procgroup File The procgroup file for a network of workstations contains a separate line for each process to start. More than one process may be run on one machine, however, each process must be specified on its own line. The procgroup file for one process looks as follows.

```
local 0
```

For two processes it looks like:

```
local 0
shark 1 /home/williams/PERFECT/bin/sun4/seis
```

This will run one process on the machine the user is logged into (local "0"), and one process on "shark". If the user is logged onto "shark" this procgroup file would run two processes on "shark" which would then time share the single physical processor.

An example procgroup to start eight processes is:

```
local 0
juju 1 /home/williams/PERFECT/bin/sun4/seis
vulcan 1 /home/williams/PERFECT/bin/sun4/seis
dalek 1 /home/williams/PERFECT/bin/sun4/seis
jadoube 1 /home/williams/PERFECT/bin/sun4/seis
clone 1 /home/williams/PERFECT/bin/sun4/seis
cosmo 1 /home/williams/PERFECT/bin/sun4/seis
chartres 1 /home/williams/PERFECT/bin/sun4/seis
```

Execution To execute on a workstation network, the program must be started from the local machine. All that is required is the name of the executable. The procgroup file specifies the number and location of other processors. To run the program with eight processors use:

```
% /home/williams/PERFECT/bin/sun4/seis -pg sun8.pg
```

Special Notes If using the `alog` to produce files suitable for visualization with `upshot` on any architecture, use a workstation such as a Sun to run both the `mergelogs` and `adjlogs` commands.

Current Status May 24, 1993: `fdmod` compiles, links, and runs.
May 24, 1993: `seis` compiles, links, and runs.

B.6.2 RS6000 Network

Compilation The machine-specific file `makedef.rs6000` contains the following variables,

```
#
#  ARCO Seismic Benchmarks
#  Default make rules for IBM RS6000 running AIX
#
MACHINE = RS6000
P4_HOME_DIR = /usr/local/p4-1.3/$(MACHINE)
```

```
FC = xlf
FFLAGS = $(FOPT) -qextname
CC = cc
CFLAGS = $(COPT) -D$(ARCH)
CLINKER = cc
FLINKER = xlf -bloadmap:rs6000map -bnso -bI:/lib/syscalls.exp
OTHER = -lbsd
XLIB = -lX11

COPT = -O2
FOPT = -O

AR = ar
RANLIB = true
MDEP_LIBS = -lbsd
MDEP_FFLAGS = -qextname

LIBS = $(P4_HOME_DIR)/lib_f/libp4_f.a $(P4_HOME_DIR)/lib/libp4.a
LIB = $(BENCH)/lib/$(TARGET_ARCH)/libbench.a
SLIB = $(BENCH)/lib/$(TARGET_ARCH)/libseis.a
BIN = $(BENCH)/bin/$(TARGET_ARCH)

P4_CFLAGS = -I$(P4_HOME_DIR)/include
P4_FFLAGS = -I$(P4_HOME_DIR)/lib_f
```

Procgroup File An example procgroup to start four processes is:

```
local 0
raft 1 /home/williams/PERFECT/bin/rs6000/fdmod
wherry 1 /home/williams/PERFECT/bin/rs6000/fdmod
kayak 1 /home/williams/PERFECT/bin/rs6000/fdmod
```

Execution The program must be started from the machine which is to be the local node. To run with four processors type:

```
% /home/williams/PERFECT/bin/rs6000/fdmod -pg rs60004.pg
```

Special Notes Running on the RS6000's is very similar to a network of Sun's. Before running anything on the IBM SP-1, first run on a newtork of RS6000's, as they are binary compatable. Then copy the executables to the SP-1 directory and continue from there.

Current Status May 19, 1993: `fdmod` compiles, links, and runs.
May 19, 1993: `seis` compiles, links, and runs.

B.7 Shared Memory Machines

`P4` uses either of two methods to start processes for message passing on shared memory machines: shared memory or unix sockets. Although unix sockets are generally slower, either method may be used as specified in the procgroup file. If all processes are specified on one line, then shared memory is used. If all processes are specified on separate lines, unix sockets are used.

B.7.1 KSR-1

The KSR-1 is a Non-Uniform Memory Access machine with the memory physically distributed, but treated as shared. Nodes are connected in uni-dimensional rings of 32 which are then stacked to achieve larger numbers of processors. All memory is stored in the individual "caches" in each node, no "absolute" memory addresses exist.

Compilation The machine-specific file `makedef.ksr` contains the following:

```
MACHINE = KSR
P4_HOME_DIR = /usr/lusk/p4-1.3

LIBS = $(P4_HOME_DIR)/lib_f/libp4_f.a $(P4_HOME_DIR)/lib/libp4.a
LIB = $(BENCH)/lib/$(TARGET_ARCH)/libbench.a
SLIB = $(BENCH)/lib/$(TARGET_ARCH)/libseis.a
BIN = $(BENCH)/bin/$(TARGET_ARCH)

P4_CFLAGS = -I$(P4_HOME_DIR)/include
P4_FFLAGS = -I$(P4_HOME_DIR)/lib_f

XLIB = -lX11
RAN = ranlib

CFLAGS = $(COPT) -v -V -xfpu3 -para
FFLAGS = $(FOPT) -v -V -xfpu3 -i4 -r8 -para
FOPT = -O2
COPT = -O2

.c.a:
        $(COMPILE.c) -o $% $<
        $(AR) $(ARFLAGS) $@ $%
```

```
        $(RM) $%

.f.a:
        $(COMPILE.f) -o $% $<
        $(AR) $(ARFLAGS) $@ $%
        $(RM) $%
```

Procgroup File To use shared memory the procgroup file for the KSR-1 is the string "local" and the number of slaves. The following procgroup starts one process using shared memory.

```
local 0
```

The procgroup below starts 16 processes using shared memory:

```
local 15
```

To start processes using UNIX sockets, the procgroup file would have a separate line for each process. The following procgroup would start four processes using UNIX sockets.

```
local 0
ksr2 1 /u1/williams/PERFECT/bin/ksr/fdmod
ksr2 1 /u1/williams/PERFECT/bin/ksr/fdmod
ksr2 1 /u1/williams/PERFECT/bin/ksr/fdmod
```

Currently, the socket method does not work with the KSR.

Execution Running on the KSR-1 is very simple. The procgroup file designates the number of processors needed and allocates them. To run, type the executable name and pass any arguments needed to `p4`. Assuming an executable named `fdmod` in the `bin` directory, and a procgroup named `ksr16.pg` specifiying 16 total processes, the command to run would be,

```
% /home/williams/PERFECT/bin/fdmod -pg ksr16.pg
```

Special Notes The KSR creates a program called `trace.out` when an executable crashes. If `fdmod` failed, the problem could be examined in the following manner,

```
% stacktrace fdmod trace.out
```

A special problem with the KSR involves the default size of integers, which is four bytes in C and eight bytes in Fortran 77. Since the Fortran calls to `p4` are really just an interface to C functions, the differences in size prove fatal. The `-i4` shown above sets the default size of an integer to four bytes in Fortran. The `-r8` performs a similar function, by setting the default size for double precision to eight bytes.

Current Status May 21, 1993: `fdmod` runs, but never exits. Ming mentioned that destruction of threads can take up to 900 seconds, so I might not be patient enough. Will have to check up on that.
May 21, 1993: `seis` will not run. It crashes early, something to do with the seismic trace files.

B.7.2 Alliant FX/2800

The Alliant is a shared memory machine. It supports both unix sockets and shared memory for inter-process communication (see Section B.7. The shared memory works the best, but either may be used as determined by the procgroup.

Compilation The machine-specific file `makedef.fx2800` contains the following variables,

```
MACHINE = FX2800
P4_HOME_DIR = /fx2800/usr8/lusk/p4-1.3

FC = fortran
FFLAGS = $(FOPT)
CFLAGS = $(COPT) -D$(ARCH)
FOPT = -Ogvc
COPT = -O
COMPILE.f = $(FC) $(FFLAGS) -c
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) -c
FLINKER = $(FC)
#
#  add ranlib to make rules for libraries

AR = ar
ARFLAGS = rv
RAN = ranlib
RM = rm

LIBS = $(P4_HOME_DIR)/lib_f/libp4_f.a $(P4_HOME_DIR)/lib/libp4.a
LIB = $(BENCH)/lib/$(TARGET_ARCH)/libbench.a
SLIB = $(BENCH)/lib/$(TARGET_ARCH)/libseis.a
BIN = $(BENCH)/bin/$(TARGET_ARCH)

P4_CFLAGS = $(P4_HOME_DIR)/include
P4_FFLAGS = $(P4_HOME_DIR)/lib_f

.c.a:
```

```
        $(COMPILE.c) $<
        $(AR) $(ARFLAGS) $@ $%
        $(RAN) $@
        $(RM) $%

.f.a:
        $(COMPILE.f) $<
        $(AR) $(ARFLAGS) $@ $%
        $(RAN) $@
        $(RM) $%
```

Procgroup File See Section B.7 for a description of the posibilities for a procgroup on shared memory machines. For one process, the procgroup would look like this:

```
local 0
```

The following is the procgroup for two and then eight processes using shared memory, and then two and eight using sockets.

```
local 1 /usr8/levine/PERFECT/bin/fx2800/fdmod

local 7 /usr8/levine/PERFECT/bin/fx2800/fdmod

local 0
hurley 1 /usr8/levine/PERFECT/bin/fx2800/fdmod

local 0
hurley 1 /usr8/levine/PERFECT/bin/fx2800/fdmod
hurley 1 /usr8/levine/PERFECT/bin/fx2800/fdmod
hurley 1 /usr8/levine/PERFECT/bin/fx2800/fdmod
hurley 1 /usr8/levine/PERFECT/bin/fx2800/fdmod
hurley 1 /usr8/levine/PERFECT/bin/fx2800/fdmod
hurley 1 /usr8/levine/PERFECT/bin/fx2800/fdmod
hurley 1 /usr8/levine/PERFECT/bin/fx2800/fdmod
```

Execution Give the filename and any options to run (such as the procgroup name.) No explicit acquisition of nodes is required. E.g.,

```
% /usr8/levine/PERFECT/bin/fx2800/fdmod -pg fx28002.pg
```

Special Notes On the FX/2800 we used, the memory requirements were too large. Add a switch to increase the amount of global memory used.

```
% /usr8/levine/PERFECT/bin/fx2800/fdmod -pg fx28002.pg -gm 2000000
```

Also, for `fdmod`, the comment lines in `parmfile` caused an error. Delete the first three lines and modify `getparm.f` by commenting out the first three reads.

Current Status As Alliant has gone out of business, providing full functionality on the FX/2800 has not been a high priority. `fdmod` has been run, and `seis` should be a relatively port.
May 20, 1993: `fdmod` ran earlier, but now seems to crash with two processors.
May 20, 1993: A minimal effort compiled `seis`, but it doesn't run. No further effort was put there.

B.7.3 BBN TC2000

As a shared memory machine, the TC2000 may use sockets or shared memory. (See the discussion under Section B.7.)

Compilation The machine-specific file `makedef.tc2000` contains the following variables,

```
MACHINE = TC_2000
P4_HOME_DIR = /usr/local/$(MACHINE)

FC = f77
CC = cc

RAN = ranlib
AR = ar
ARFLAGS = r

CFLAGS = $(COPT)
FFLAGS = $(FOPT) -par
COPT =
FOPT =
FLINKER = $(FC)

LIBS = $(P4_HOME_DIR)/lib_f/libp4_f.a $(P4_HOME_DIR)/lib/libp4.a
LIB = $(BENCH)/lib/$(TARGET_ARCH)/libbench.a
SLIB = $(BENCH)/lib/$(TARGET_ARCH)/libseis.a
BIN = $(BENCH)/bin/$(TARGET_ARCH)

P4_CFLAGS = -I$(P4_HOME_DIR)/include
P4_FFLAGS = -I$(P4_HOME_DIR)/lib_f
```

```
.c.a:
        $(COMPILE.c) -o $% $<
        $(AR) $(ARFLAGS) $@ $%
        $(RAN) $@
        $(RM) $%

.f.a:
        $(COMPILE.f) -o $% $<
        $(AR) $(ARFLAGS) $@ $%
        $(RAN) $@
        $(RM) $%
```

Procgroup File Examples of one, two, and eight processors will be shown, first for shared memory and then for sockets.

```
local 0

local 1 /home/williams/PERFECT/bin/tc2000/fdmod

local 7 /home/williams/PERFECT/bin/tc2000/fdmod

local 0

local 0
lepido 1 /home/williams/PERFECT/bin/tc2000/fdmod

local 0
lepido 1 /home/williams/PERFECT/bin/tc2000/fdmod
lepido 1 /home/williams/PERFECT/bin/tc2000/fdmod
lepido 1 /home/williams/PERFECT/bin/tc2000/fdmod
lepido 1 /home/williams/PERFECT/bin/tc2000/fdmod
lepido 1 /home/williams/PERFECT/bin/tc2000/fdmod
lepido 1 /home/williams/PERFECT/bin/tc2000/fdmod
lepido 1 /home/williams/PERFECT/bin/tc2000/fdmod
```

Execution Run with the cluster command to limit the number of nodes used.

```
% cluster 8 fdmod -pg tc20008.pg
```

Special Notes

Current Status BBN's advanced computer group is no longer making the TC2000 and only the port of `fdmod` has been attempted. The port to `seis` should be easily accomplished.
May 19, 1993: `fdmod` compiles and runs.
May 19, 1993: `seis` cannot be run until the new version of p4 supports it. Will `p4` be ported to the TC2000?

B.7.4 Sequent Symmetry

The Symmetry has been used for only `fdmod`.

Special Notes Shared memory message passing is not supported in Fortran by p4.

Current Status

B.8 List of Common Problems

"pgm_path_name: Command not found" `P4` tried to start the program with the given name on a remote machine and the program did not exist. Verify the full path name of the program.

"make: Warning: Can't find '/include/makedef.-sun4' Environment variables are not set. Set `BENCH`, `ARCH`, and `TARGET_ARCH` before trying to make or run.

"SEIOPEN: ERROR OPENING path/stest1.HDR" `Small1.prm` must be used to create stest1.* before `small2.prm` is run.

B.9 Results

"these results are not officially approved and reported by the SPEC Perfect Group Steering Committee. They may not be directly comparable to accepted and verified results."

Table 3: FDMOD timing results

Machine	1 (50)	2 (70)	4 (80)	8 (110)	16 (130)
Sun	3.08019	7.63494	7.20896	30.8019	
nCUBE	4.25984	5.76716	4.32537	8.65075	6.02931
iPSC/860	1.83700	2.30100	1.75500	2.36000	
DELTA	1.64100	2.21900	1.69300	2.35100	2.23900
KSR - SM	2.73999	3.96000	3.31999	4.29999	4.01999
KSR - Sock					
TC-2000 - SM	28.785	75.536	86.546	104.41	103.019
TC-2000 - Sock	28.785	97.297	115.75	392.09	1050.85
FX/2800 - SM	4.56	5.68	4.95	8.51	8.07
FX/2800 - Sock	4.56	6.98	8.97		
Symmetry - Sock	42.86	116.050			
CM-5					
RS6000	3.96799	7.03999	12.2880	34.1759	
SP-1	.639999	1.15199	2.55999	31.2320	42.3680

Table 4: SEIS timing results

Machine	1 (S1)	1 (S2)	2 (S1)	2 (S2)	4 (S1)	4 (S2)	8 (S1)	8 (S2)
Sun	1.20.1	41.47.1	57.9	43.02.8	0.47.8	18.47.0	0.47.9	10.01.1
iPSC/860	1.36.9	5.12.5	1.31.9	10.45.5	2.41.9	8.12.1		
DELTA	1.29.5	4.14.0	1.29.8	3.40.8	0.46.0	4.42.8	1.02.2	4.45.0
RS6000	2.42.0	2.14.4	1.20.8	1.11.9	1.15.1	1.26.6	0.55.5	1.40.2
SP-1	0.26.6	0.20.9	0.20.1	0.19.4	0.27.8	0.24.0	0.32.8	0.43.7

References

[1] Kuck, D. J., and A. H. Sameh, *A Supercomputing Performance Evaluation Plan*, Tech. Rep. 692, Center for Supercomputing Research and Development, University of Illinois, June 1987.

[2] Berry, M., et al., *The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers,* The International Journal of Supercomputer Applications, 3 (1989), pp. 5–40.

[3] Mosher, C. C., S. Hassanzadeh, and F. G. Lou, *ARCO Seismic Benchmark Users Guide*, Version 1.00, January 1992.

[4] Ralph Butler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, Mathematics and Computer Science Division, October 1992.

[5] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with Upshot. Technical Report ANL-91/15, Argonne National Laboratory, Mathematics and Computer Science Division, August 1991.