Chapter 6

A Simulation Tool for MPI Performance Prediction

Reverse profiling exhibits many of the characteristics of simulation in that each process maintains its own simulation clock. However it sidesteps the synchronisation problems of parallel simulation by requiring that all communications are deterministic. This means that programs using wildcarded receives (i.e. "receive the first message to arrive from any other process") cannot be handled accurately using reverse profiling; reverse profiling will select the first (in real time) message to arrive rather than the first (in simulation time) to arrive.

Discrete event simulation is needed to handle this non deterministic case properly. Indeed simulation has been suggested as a cost effective method for developing and debugging parallel programs. Models may be as complex as desired to incorporate the detailed behaviour of the hardware.

This chapter describes a simulation tool for MPI performance prediction. The tool has a standard MPI interface so programs may be moved from the simulation development platform onto the final machine with minimal effort. A comparison with actual results is presented and the ease of use of this approach is discussed.

6.1 Introduction

The point of using simulation for development (rather than developing on the parallel machine itself) is that it provides a stable *repeatable* environment. Brooks [5] highlighted the importance of having a simulator available during the development of a new machine, mentioning that the important thing was not that the simulator should be a perfect representation of the real machine, but that if there were bugs at least they would be the *same* bugs each time the program is run. Brewer [3] discusses the advantages and disadvantages of simulation for parallel

program development.

The problems with simulation are speed, accuracy and the time spent developing models. Speed may be compromised for accuracy (or vice-versa) by developing more or less detailed models. The time spent developing models is a real issue; the standard approach (in, for example the WWT [12], Proteus [2] and PS [1]) is to develop a network model of the architecture based on hardware design documents and then to refine this until predicted and measured times for a suite of programs fall into line.

The work in this chapter is based on the HASE simulation tool described in section 6.2. HASE was developed to allow modelling at any level of abstraction, from high level algorithm simulations to detailed hardware. Section 6.3 uses models at the different levels for MPI performance prediction, to assess their ease of use. Section 6.4 links HASE with the MPI performance models of chapter 3. This produces similar predictions to reverse profiling, but can handle non deterministic cases as well as deterministic ones. Section 6.5 presents some example graphs obtained using the tool and section 6.6 draws conclusions from this investigation.

The interesting question addressed in this chapter is: What is the place of simulation in the development lifecycle?

6.2 The HASE simulator

The Hierarchical Architecture Simulation Environment (HASE) was developed as a tool for modelling and simulating computer architectures at any level of abstraction. Different models at varying abstraction levels were constructed to investigate the ways in which simulation may be applied to performance prediction. The design of this tool is outlined in the sections below, and more details may be found in the references [10], [8] and [9].

6.2.1 Overall operation

HASE allows designers to explore architectural designs at different levels of abstraction through a graphical interface based on X-Windows/Motif. The results of the simulation can be seen through animation of the design drawings. The HASE tool acts as a graphical front end to SIM++ [13], a discrete event simulation extension of C++. SIM++ is used to describe the behaviour of basic components of a simulation. It provides a `sim_entity` class from which user components may be derived. Entities notionally run in parallel and may schedule messages to other entities using SIM++ library functions. The user can link icons corresponding

to entities together on screen and HASE produces the SIM++ initialisation code necessary for simulating the network. New components can be constructed by linking together standard components. Each component can be simulated at any level of abstraction. A register transfer level simulation will produce the most accurate simulation results; behavioural level simulations run more swiftly. The tool allows different parts of the simulation to run at different abstraction levels, so the user can 'zoom in' to specific parts of the design to simulate that at a low abstraction level and run the rest of the design at a high level of abstraction. Figure 6.1 shows how the parts of the system fit together. Entities are selected from a library, and joined together to form a network. To run a simulation, HASE generates the SIM++ code for the simulation, which is compiled using Jade's SIM++ compiler. The simulation executable reads parameters generated by the HASE user interface, and produces a trace file of the execution which can be used for animation and statistics within the HASE tool.

6.2.2 Internal design of HASE

Each project built using HASE has its own directory for storing the SIM++ code. This directory may be used for building and running the simulation outwith the HASE environment using command line tools like `make`, giving the full flexibility of the SIM++ programming language. Alternatively the simulation process may be controlled from the HASE front end. HASE itself was written using C++, and a project is represented within HASE by four main classes; the **entity**, the **parameter**, the **link** and the **port**.

- **Entity.** This object stores a single component (or 'entity' in SIM++ terminology). The SIM++ code defining the behaviour is held in a file which has the same name as the entity. Within the object are stored details of the entity's ports and parameters. In addition, it holds the name of the bitmap file used for display and animation.

- **Parameter.** An entity may have many parameters. Details of these are stored within HASE along with instructions for their animation.

- **Port.** An entity sends messages to other entities via 'ports'. A port has a name, an icon and position relative to the entity's icon. The simulation code for an entity is written using sends and receives to and from these ports rather than directly to and from other entities. This constraint means that reusable components may be constructed with a defined interface.

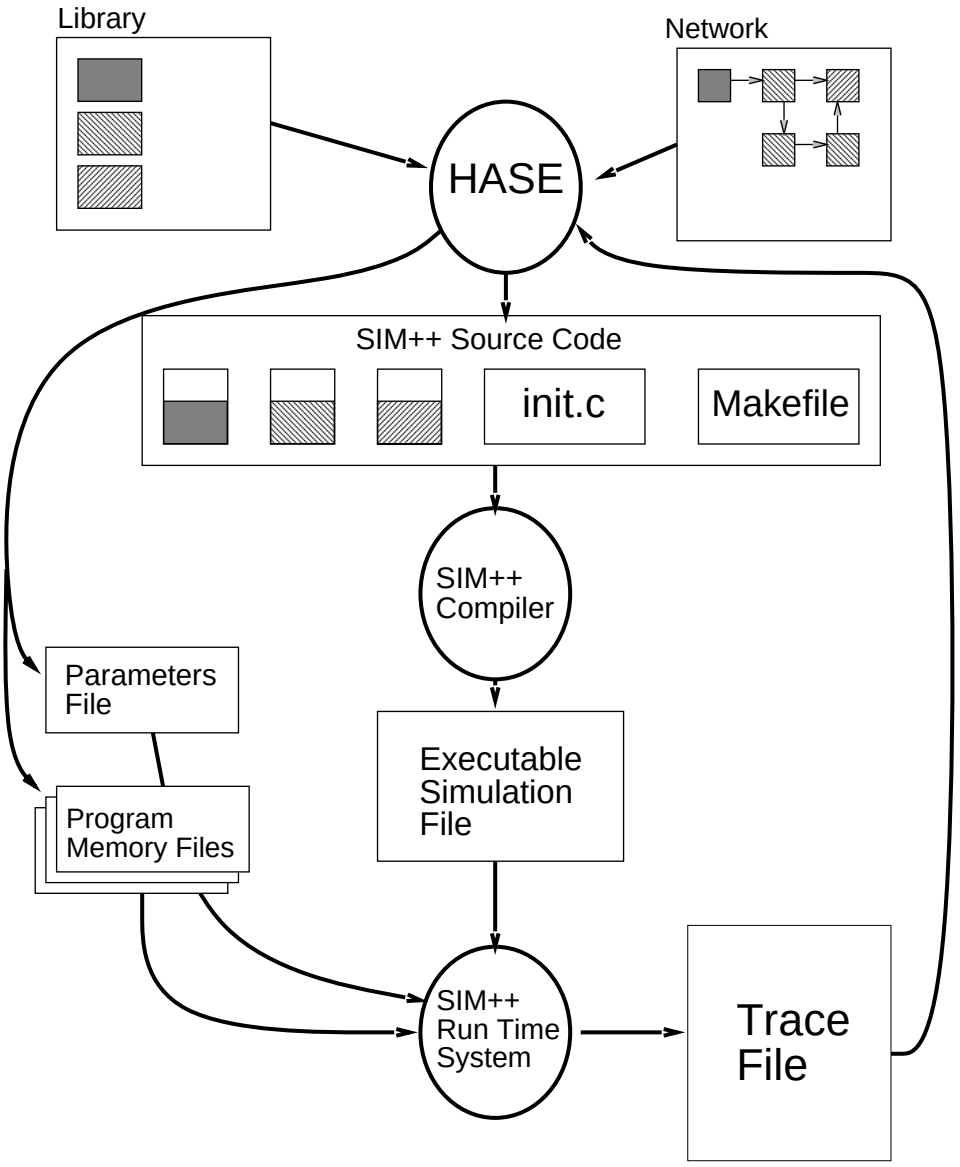

Figure 6.1: The top level design of HASE.

- **Link.** This holds a link between two ports, drawn as a line on the screen. The object includes mechanisms for animating packets sent between entities.

6.2.3 Hierarchy

A subdivided entity may be defined in terms of a network of lower level components. Sometimes this is purely to make the design more manageable on screen, with the simulation still being performed using the low level components. It is also possible to provide simulation code for this higher level component and choose to use this one object rather than the low level network in order to obtain faster simulation time and less detailed results.

This choice of simulation level may be made at run time and is made by toggling a switch associated with the object. The external interface of the high level component is defined to be the same as that of the lower level network. This allows the simulation level of each object in the simulation to be set independently. Figure 6.2 illustrates two subdivided components connected by their external ports.

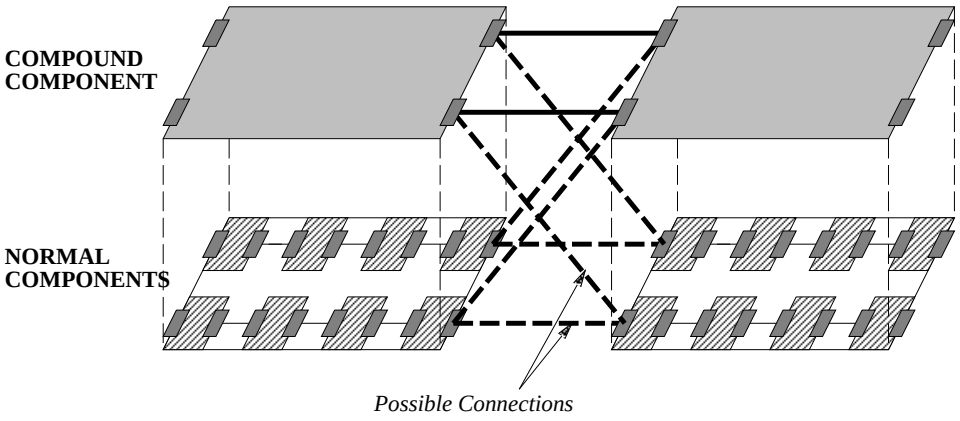

Figure 6.2: Two subdivided entities are connected by their external ports.

6.2.4 Parameter types

HASE parameters are the crucial link between the simulation code and the animation. They form the internal representation of each entity's state and include integers, floats, enums, structs and arrays. Once a parameter has been defined for an entity within HASE, that parameter is available to the simulation code as a normal C++ variable. The initial value of the parameter may be set using a Motif dialog and changes in the parameter's value may be recorded in the trace file at simulation run time, ready to be picked up by the animator. Array variables are

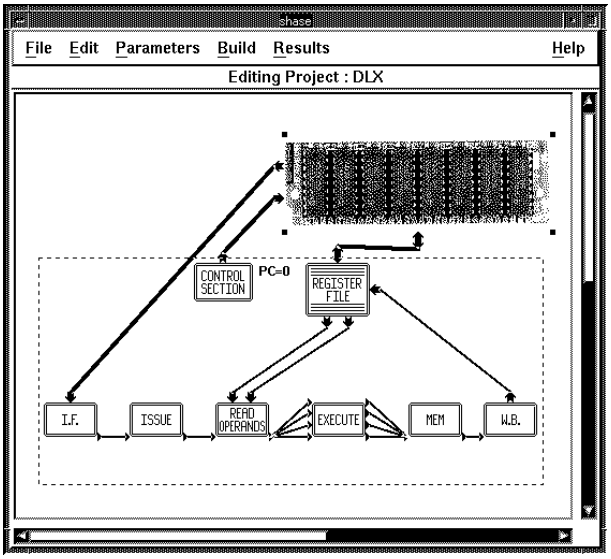

Figure 6.3: The HASE user interface.

initialised at run time by reading in a text file. This process is powerful enough to allow streams of instructions (for example consisting of `COMPUTE <time>`, `SEND <proc#>`, `RECV <proc#>`) to be parsed and read in to a component's memory.

6.2.5 Templates

Templates for building common structures such as arrays and meshes of components are included. The user can slot any component into the template, set the dimensions and all the required components and links are produced. Current templates include a linear array, a 2D mesh, an omega network and a 3D torus.

6.2.6 Output approaches

Simulations are renowned for producing vast quantities of raw data; transferring this into useful information is no trivial task. The result of a single simulation run is a trace file with timestamps showing when all changes in state and messages occurred. HASE includes two visualisation tools to make sense of this information; an animator and a timing diagram display. The hierarchy is used to control the amount of information displayed on the timing diagram and logic-analyser style measurements can be taken. Figure 6.4 shows an example display. The trace file format has three sections. The first defines the data types, the second the bars and the last the events, with time stamps. An example is :–

```
$types
State SEND RECV WAIT BUSY
```

```
Phase Init Input Calc Output
$bars
p[0] State
p[1] State
All Phase
$events
u:p[0] at 0.1234 : P ALLREDUCE
u:p[1] at 0.1254 : P SEND
```

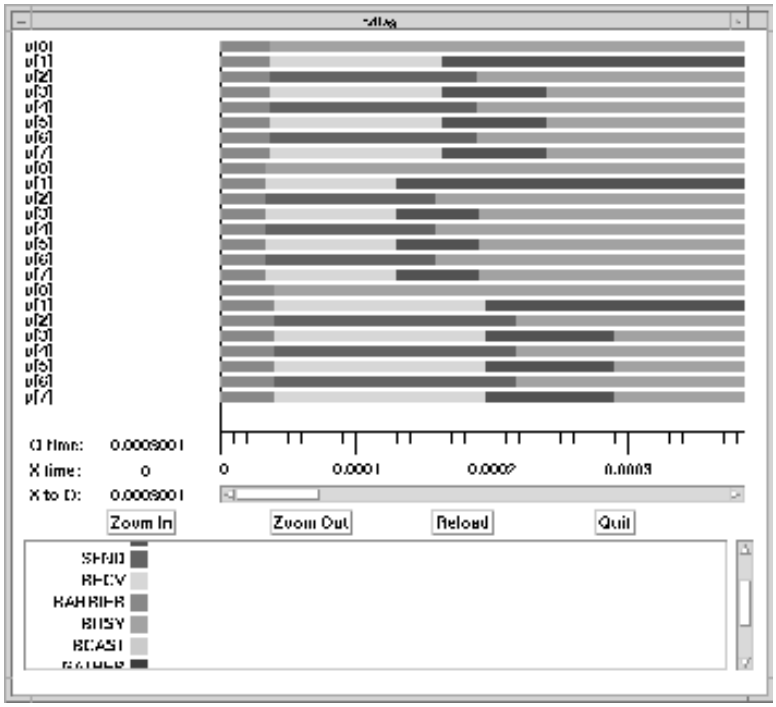

Figure 6.4: A timing diagram display.

The animator uses the trace information to show messages passing between entities as well as state changes on screen. Used in conjunction, the timing diagram and the animator show in detail what is actually going on during a simulation run, which is very useful when developing models.

For very low level debugging purposes it is sometimes necessary to resort to looking at the trace file itself. Once a model has been developed, it is natural to stretch it with heavy workloads. This can rapidly generate unmanageably large trace files, so there is a mechanism in HASE for controlling how much trace information is produced. For the largest runs it is usual to garner a small number of statistical measures from the model. These measures are taken using classes provided in SIM++ for histograms, counts and accumulated averages.

Repeated runs are required to investigate how a model behaves using a range of parameters [7]. These runs are controlled by a Perl script and graphs are produced using the GNUplot program.

6.3 Using HASE at different abstraction levels

With a simulation environment such as HASE there are no restrictions on the amount of detail which may be incorporated into a model. It is theoretically possible to simulate every piece of hardware and software of the target machine and obtain an exact prediction of the performance. In practice the simulation run times would be prohibitive and it would take a too long to build a complete simulation model. So an intermediate level must be found.

Section 6.3.1 describes work done interfacing MPI to low level network simulation models. Section 6.3.2 investigates the opposite approach - treating the parallel program as the simulation model. Section 6.3.3 describes how cycle counting may be used for accurate estimates of computation delays. Tools for debugging at the source code level are essential for making complex parallel programs (and simulations) work, so section 6.3.4 describes the facility for single stepping through simulation and MPI source code.

6.3.1 Low level models

To check how useful low level modelling can be for MPI performance prediction, the MPI interface functions were written to link with multilevel graphical models of the hardware. In addition to the performance results for the software this approach also analyses the behaviour of the underlying hardware.

Hardware models of meshes, tori, fat trees and buses were constructed with HASE. The same MPI/SIM++ interface links user code to the simulation so realistic workloads (using actual programs) may be run.

An example application was the low level implementation of an `MPI_Allreduce` with the operation of addition. The same routine was run on the different architectural models and the results animated. The animations could be run simultaneously on screen so that the architectures could be compared for this application. For this "proof of concept" experiment, the switching delays were set at 1 unit per hop and timing diagrams of the hardware and software performance were produced.

Figure 6.5 shows two different tree structures being run concurrently. The same MPI application is running on each simulation, but they take different

run times because of the different networks. The numbers below the processors show the intermediate values of computations. Figures 6.6 and 6.7 show the timing diagrams for the binary tree and the fat tree respectively. The fat tree network completes the algorithm in approximately half the time of the binary tree. Figures 6.8, 6.9 and 6.10 show the displays for a 3D torus network, a 2D torus network and an omega network respectively. Only the routing model distinguishes the models; the same MPI code runs on all structures.

6.3.2 High level models

Many graphical CAD and CASE tools have been proposed to attack the complexity of parallel programming. Few are used in practice. This is partly practical – most tools have been built as university research projects rather than as commercial applications, but also because the tools do not scale beyond toy applications.

Some simple experiments were run using HASE as a graphical CASE tool to see if the features designed for hardware animation could be applied to software animation.

For example, figure 6.11 shows a simulation of a task farm at the process level (implemented with point to point links). The number of workers may be varied and effects such as starvation may be observed. Such models are useful for illustrating certain effects (such as starvation) and exploring the limits of algorithms. However it is not clear that such modelling is applicable to "everyday" program design, where the aim is to have simple regular structures, and use collective communications in preference to the more fiddly point to point methods.

Contrasts may be drawn with parallel software engineering techniques; for example PARSE [11] uses a similar notation to that used for the task farm (i.e. processes in bubbles with named and typed ports for communication). The earlier dataflow diagram techniques of Yourdon and DeMarco for sequential software design are also similar.

However all these are *notations* rather than simulation systems, and their end product is a set of diagrams on paper rather than a working model.

Their model of communicating processes is eminently suited to small scale distributed systems, with several different types of independent processes communicating. It is less applicable to a parallel program written as a sequence of operations on distributed data structures (the diagram reduces to a single circle, or a set of identical circles, and yields little information).

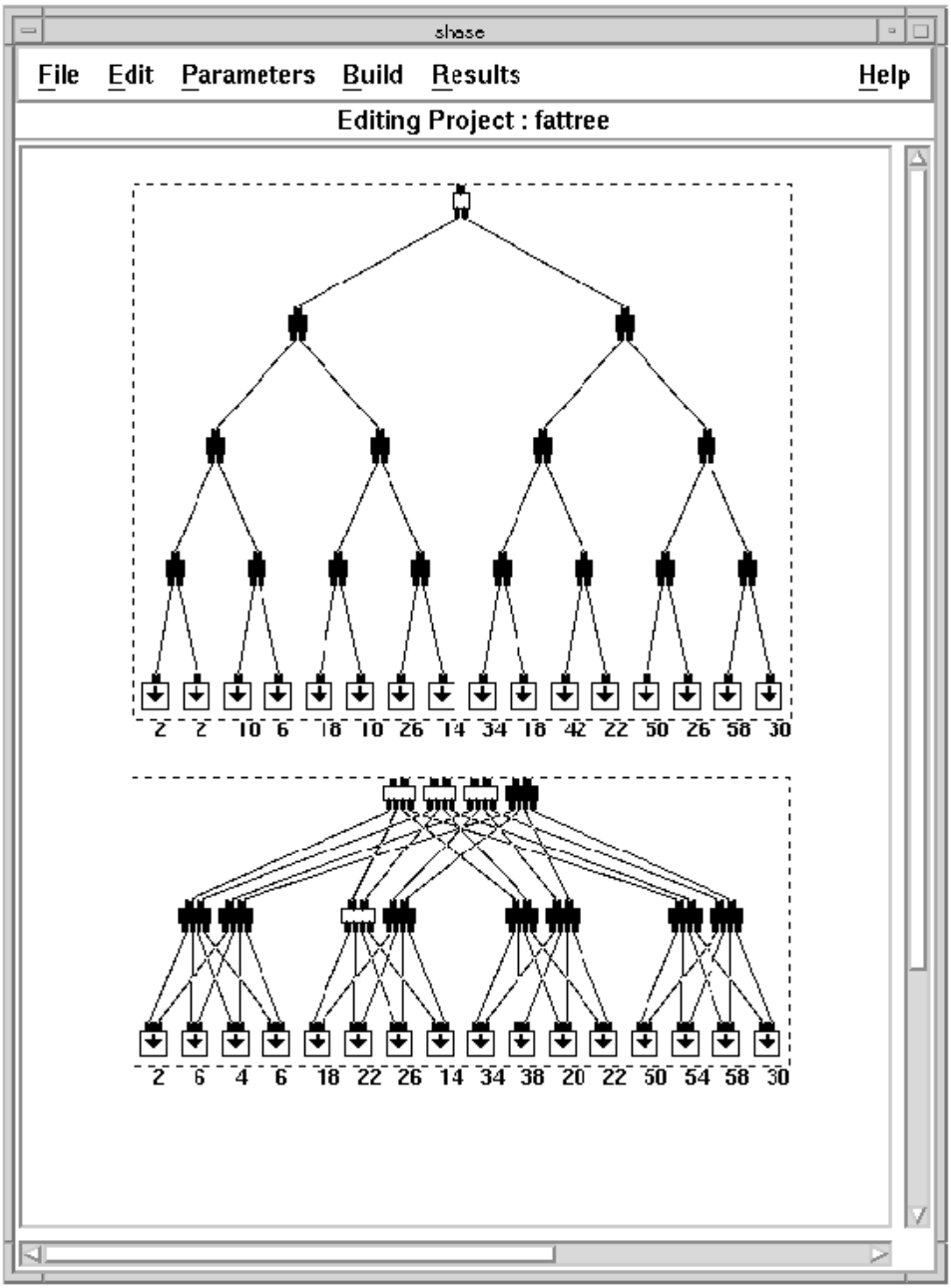

Figure 6.5: A graphical representation of two architectures; a binary tree (top) and a fat tree (bottom).

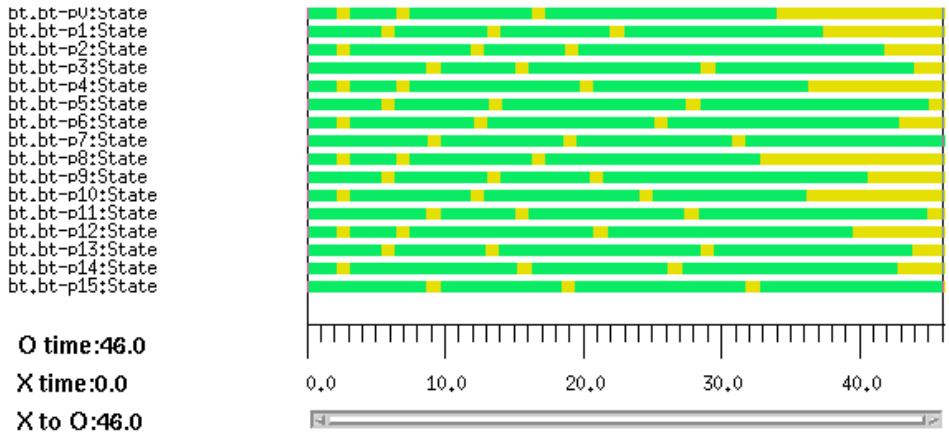

Figure 6.6: A timing diagram showing the detailed behaviour of a binary tree implementing an **allreduce** communications operation.

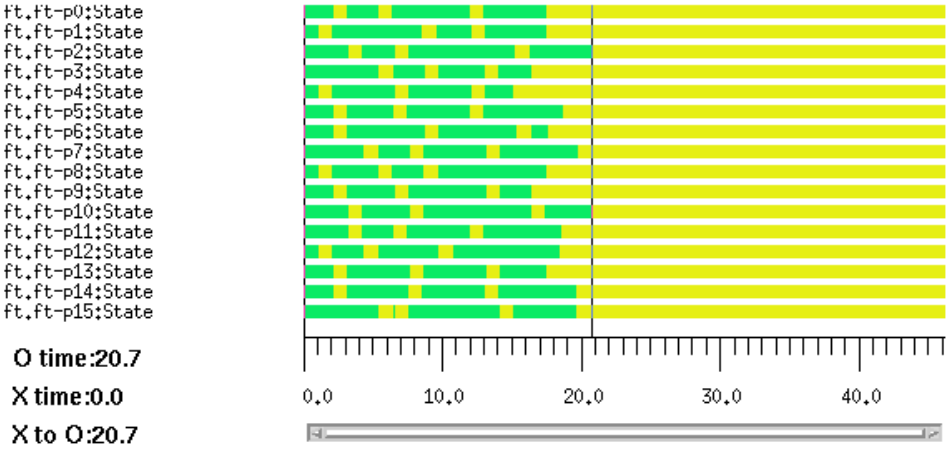

Figure 6.7: A timing diagram showing the detailed behaviour of a fat tree implementing an **allreduce** communications operation.

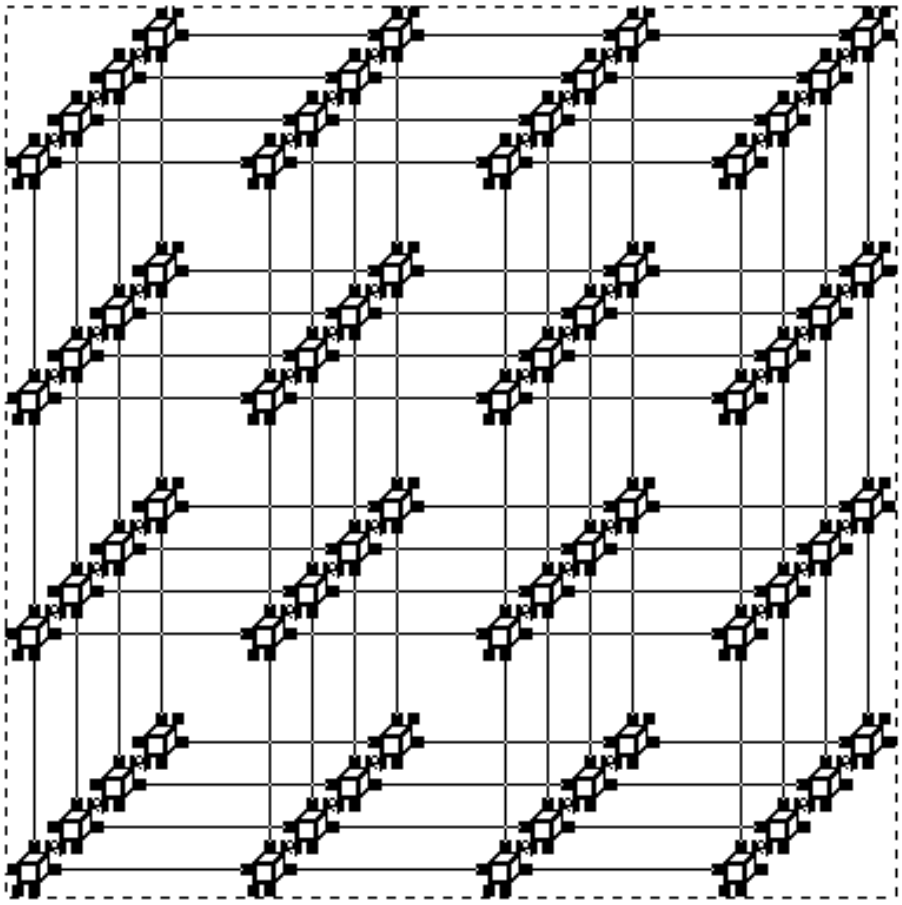

Figure 6.8: A HASE simulation of a 3d torus.

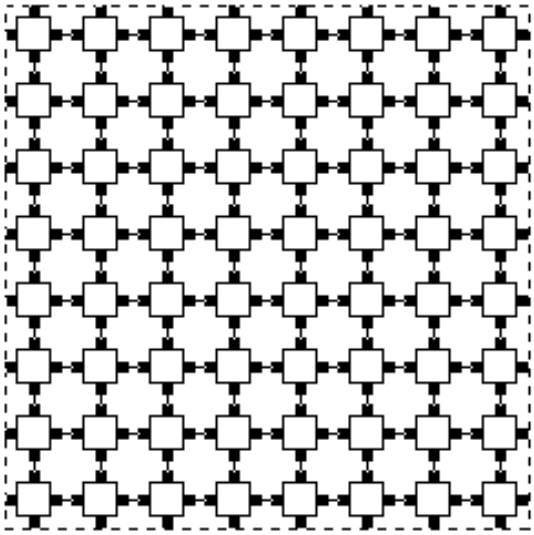

Figure 6.9: A HASE simulation of a 2d torus.

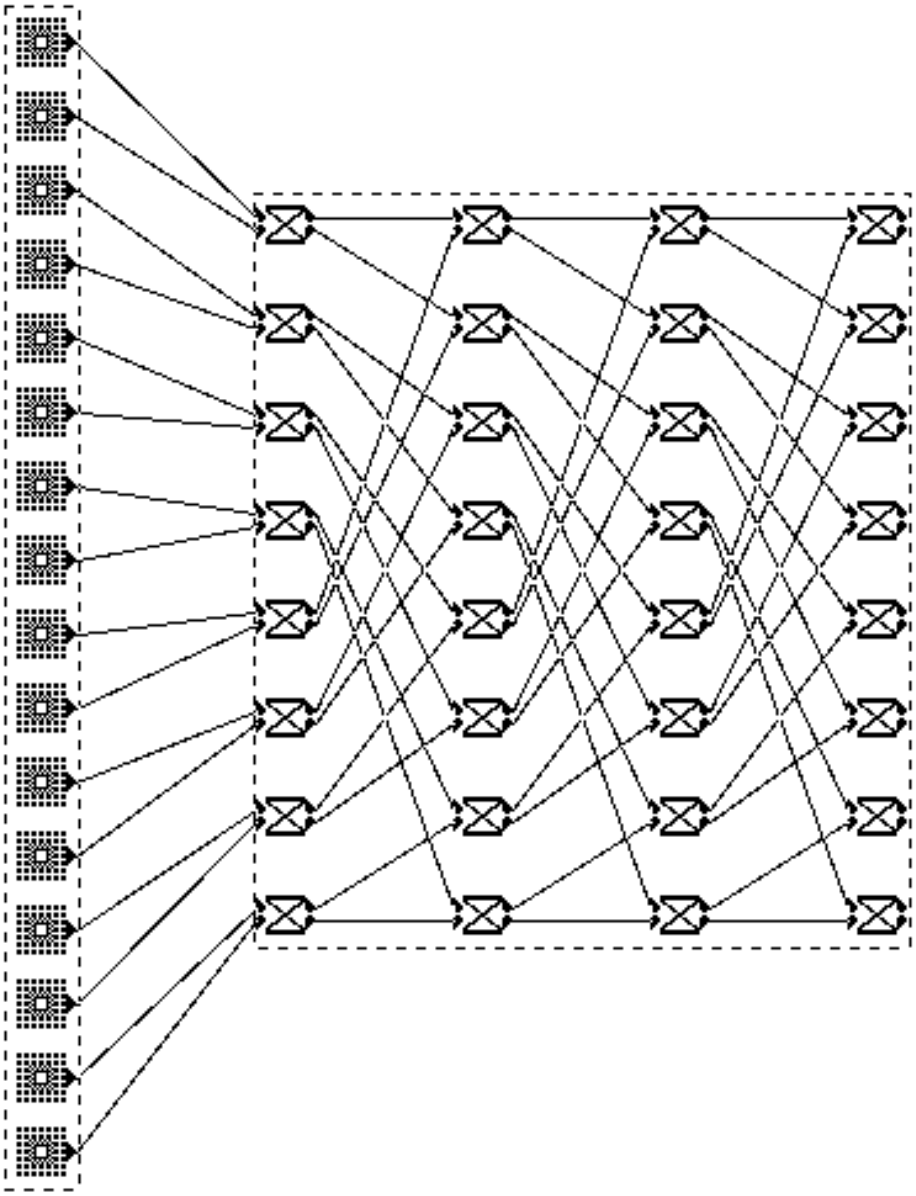

Figure 6.10: A HASE simulation of an omega network.

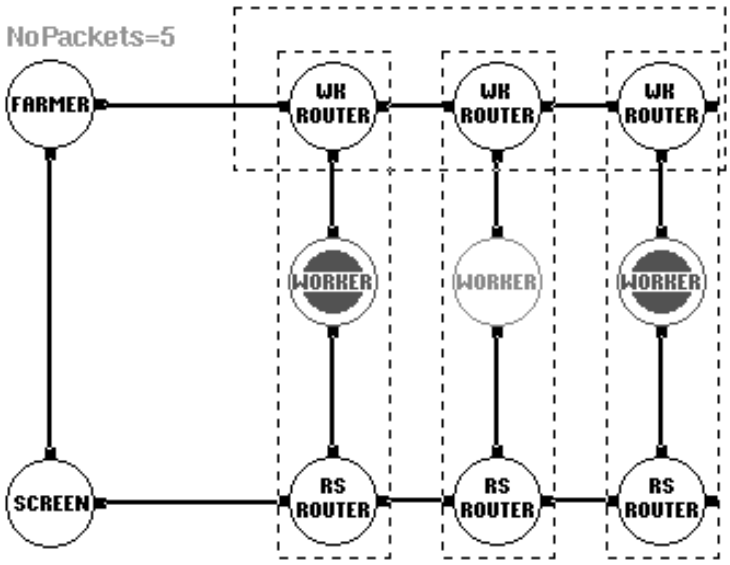

Figure 6.11: A task farm.

6.3.3 Cycle counting

In addition to simulating the performance of the MPI functions it is necessary to determine the speed of the computation. There are various methods of doing this; estimating from the source code, letting the user guess delays, full instruction set simulation or cycle counting. This last option involves processing the assembler and adding instructions to update a cycle count to each basic block. This can give accurate results [2] at the cost of slowing down execution by a factor of two.

To investigate the applicability of cycle counting, the code augmenter from Proteus was used to add cycle counting to SIM++ simulations.

Some modifications to the Proteus augmenter were required for Solaris, but it worked effectively. To be realistic *all* code must be augmented (including standard I/O and maths libraries) which is a systems administration burden since libraries like `libc.a`, `libm.a` must be recompiled from source (which is not always available). The augmenter must also support the target architecture for realistic results.

In conclusion, cycle counting does work, but requires a fair amount of effort (and access to the source code of all libraries used).

A simpler alternative is just to time the intervals between communications and scale the times up or down by the amount the target processor is slower or faster than the development processor. This is made tricky in a multi threaded implementation as there is usually only one clock which will measure the wall clock

time spent while other threads are running so the times become meaningless.

6.3.4 Single stepping

A major criticism of parallel tuning tools is that there is poor linkage between the displays and the original source code (no such facility appears in the ParaGraph tool for example). Such features have been incorporated into recent tools such as the IBM RP/2 system. The feature was added to HASE to highlight the source line of each object to allow single stepping. Figure 6.12 demonstrates this highlighting for two of the MPI processes running on a torus network.

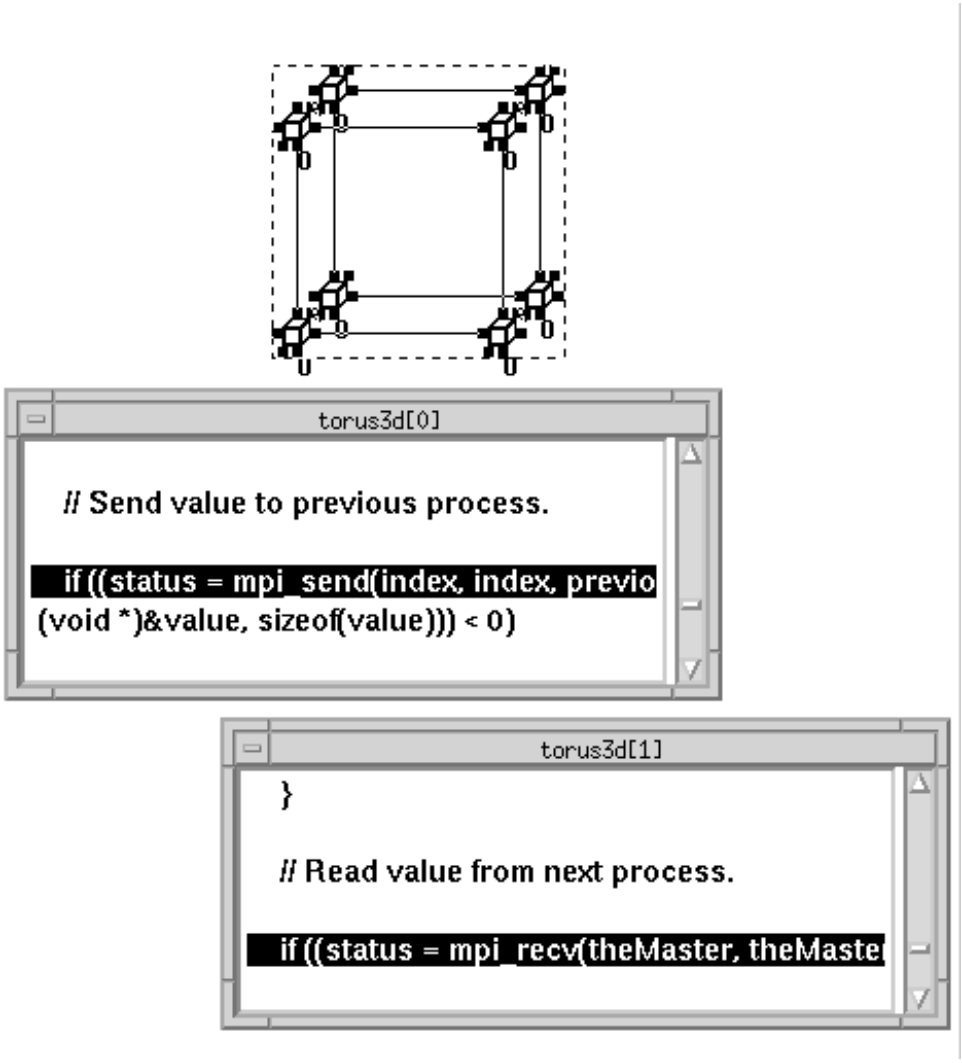

Figure 6.12: Source line highlighting.

This has proved useful for SIM++ code development and also for demonstrating simple MPI algorithms. Such debugging may be added to a simulator without affecting the behaviour (unlike a profiler).

6.4 Using HASE with MPI performance models

This section describes the use of HASE with the MPI performance models of chapter 3. This technique required a re-implementation of the MPI functions written using the simulation primitives. User code is linked with the simulation code, and the simulation runs concurrently with the user's application to produce a predicted trace file.

The technique produces the same results as reverse profiling for deterministic applications. But since the simulator keeps track of global simulation time, non-deterministic applications may also be handled correctly.

6.4.1 Implementation

The simulator is based on SIM++ [13] which extends C++ to include lightweight processes and events. The unit of concurrency is a parallel "object" which maps to the MPI model neatly. SIM++ provides a more powerful parallel programming model than MPI since shared variables are allowed in addition to message passing. It also incorporates the notion of time to schedule the objects.

Each MPI process is allocated a separate object and each runs in parallel. MPI function calls are intercepted by methods local to the object and are implemented in terms of the SIM++ primitives:

```
void sim_schedule(...);
void sim_wait( sim_event &ev );
void sim_hold( sim_time delay );
```

Thus there is a `class process` which looks like:

```
class process : public sim_entity {
public:
  int MPI_Send(...);
  int MPI_Recv(...):
  int MPI_Barrier(...);

  void body();
}
```

The `body()` method performs the actual work.

```
void process::body()
{
```

```
  // The user's main() routine goes here
  // All MPI calls are intercepted either
  // by local methods or by global functions MPI_Init
}
```

The simulated versions of MPI routines are implemented using SIM++. For example the implementation of `MPI_Send(...)` uses a `sim_schedule(...)` to send the message, along with a `sim_hold(...)` to delay for the expected delay. `MPI_Recv(...)` is implemented using `sim_wait(...)`.

The collective MPI routines are implemented using the point to point functions, with the delays calculated using the collective models rather than the constituent point to point models.

6.4.2 Implications of a threaded model

There are several implications of a threaded model as opposed to the usual process model for MPI (although the MPI standard [6] does not specify the process execution model and does mention the possibility of a shared memory implementation). The main difference is the treatment of global variables; these are private in a process based implementation and shared with SIM++. Private variables can be included in a SIM++ implementation by making them members of the class.

To summarise; care must be taken with global variables when moving code from a process based MPI to a threads based MPI.

There is also an issue when using C code with SIM++. Most of the MPI calls are implemented using methods of the process object to which C code does not have access. This problem has been solved for SIM++ by defining C wrapper functions which call the SIM++ version for the current object. For example:

```
int MPI_Send(...)
{
  return current_process->MPI_Send(...);
}
```

The final issue concerns the `main()` function itself. This has to be renamed as `mpi_main()` in order to link correctly.

Thus, with minor modifications, any C or C++ MPI code may be run on the prototype SIM++ run time system. A production simulation environment could negate the need for even the minor changes.

6.4.3 The performance model

The time delays are calculated according to a model derived from the routines of chapter 3. Thus the times could also be calculated by hand using the published table of the model. This is important, as simulation is intended to be *one of the* development tools rather than a utopian solution, so it is essential that the logic which arrived at timings may be checked by hand. (rather than a "black box" simulator mysteriously generating times which can't be checked without delving into its dark recesses).

6.4.4 A FORTRAN linkage model

Emphasis has been placed on linking C and C++ so far. However most scientific users use FORTRAN, so it would be beneficial if a development method could support FORTRAN as well as C users.

At first sight the problem appears straightforward; compile the Fortran routines separately and link them with SIM++. However there are several obstacles. The route from Fortran to C is well trodden, but that between Fortran and SIM++ is less well known. The main differences are:

- Arrays; C orders them row major in memory whereas in Fortran they are column major.
- Function arguments; in C they are passed by value, whereas in Fortran they are passed by reference.
- Naming; Fortran functions are preceded with an underscore.
- Globals; This is a more serious issue with Fortran than with C as global variables cannot be moved into the class to make them inaccessible to other processes.
- Static variables; This is a serious problem with Fortran. Local variables are the equivalent to `static` variables in C. This means that multiple threads calling a function will update the same instantiation of a variable rather than independent copies which understandably causes havoc. Short of replacing all static variables with dynamic ones (or with function parameters which are actually local) there is no way around this problem. Note that this also make recursion awkward in Fortran.

There is also the same issue of calling the C++ methods as there was from C, so the route from Fortran to SIM++ goes via a C function.

An alternative approach would be to use a language translator (such as `f2c`). However this makes the compilation phase slower so is undesirable.

To test the Fortran route, some of the Genesis benchmark suite codes [14] were linked to the SIM++ implementation of MPI. These included the FFT1 and the QCD1 codes. The task was fairly time consuming as the codes had first to be moved from PARMACS to MPI, but could be done. The big problem was the storage class of local function variables in Fortran (they are declared static); in other words if several threads call the same Fortran function they each update the same copies of the local variables.

No simple solution to this was found. Fortran could not be made to marry well with a threaded C++ simulator without excessive source code modifications and attention was shifted to concentrate on C/C++ development instead. Fortran has been targetted with a parallel simulator in the LAPSE project [4] which uses separate processes on an Intel message passing library rather than separate threads. Reverse profiling also works happily with Fortran.

6.4.5 Speed of SIM++/MPI vs LAM/MPI

An experiment was conducted to compare the run time of MPI programs compiled with a standard distribution of MPI (LAM) and the same program running on top of SIM++.

The program for the test was a simple pingpong:

```
/* ping pong */
int b;
for (int j=0; j<1000; j++) {
  if ((rank&1)==0) {
    MPI_Send(&rank,1,MPI_INT,rank+1,100,MPI_COMM_WORLD);
    MPI_Recv(&b,1,MPI_INT,rank+1,100,MPI_COMM_WORLD,&st);
  }
  else {
    MPI_Recv(&b,1,MPI_INT,rank-1,100,MPI_COMM_WORLD,&st);
    MPI_Send(&rank,1,MPI_INT,rank-1,100,MPI_COMM_WORLD);
  }
}
```

with each process bouncing messages to and from its neighbour. This was run on a single workstation with from 2 to 32 processes. Each measurement is the average of three repeated runs; timings were taken from the same Sun

Sparcstation; no compiler flags were switched on; timings were taken using the Unix `time` command. LAM version 5.2 and SIM++ version 3.10 were used.

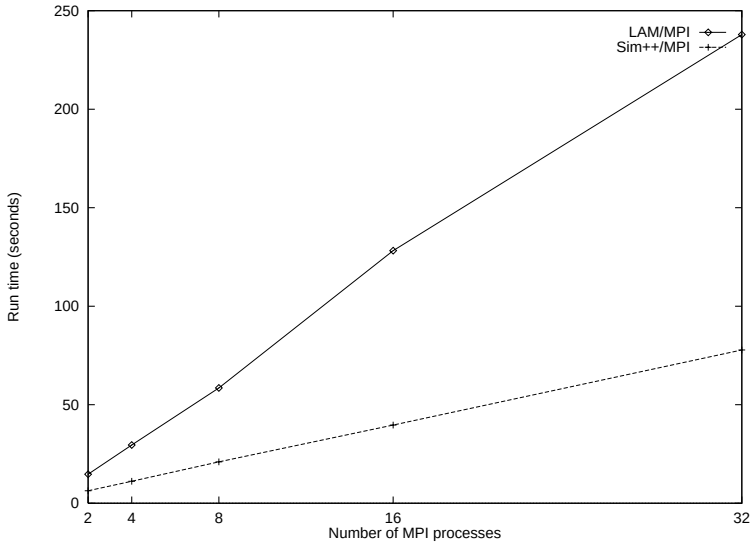

Figure 6.13: Pingpong run times on LAM/MPI and Sim++/MPI

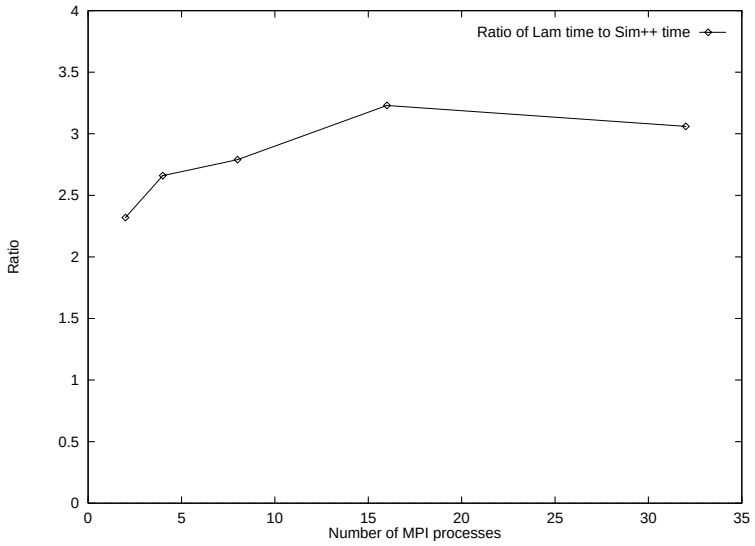

Figure 6.14: Ratio of LAM/MPI to Sim++/MPI run time

The results are shown in (figure 6.13). The SIM++ implementation of MPI is 2 to 3 times faster than LAM (figure 6.14), even though the simulation has the overhead of calculating the expected times for each communication. The reason for the apparently anomalous result of the simulation being faster than the real thing lies in the underlying implementations of LAM and Sim++. LAM gives a separate Unix process to each MPI process, whereas in Sim++ lightweight threads are used instead. The overheads of Unix process context switching are more than enough to counteract the extra burden of computing times in Sim++.

Compute intensive programs are another story. Sim++ is sequential, so the simulation time grows linearly with the number of processors simulated.

6.5 Examples

6.5.1 Cowichan problems

All the Cowichan problems were written to use deterministic patterns of communication, so the predictions obtained using the simulation tool were identical to those produced using reverse profiling (but were more difficult to obtain).

For example, figure 6.15 shows a predicted timing diagram for the `mandel` routine on eight processors of the Cray T3D, which may be compared to the reverse profiling measurements and predictions of section 5.4.1. This example required source code modifications to link it with the simulator, and took 77*s* to run on a workstation.

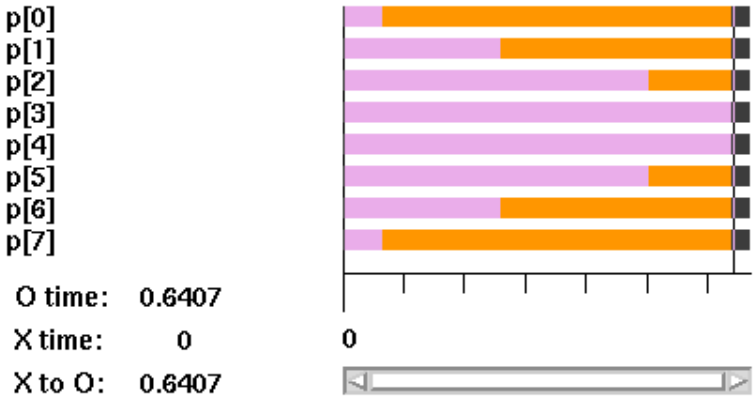

Figure 6.15: Predicted timing diagram for the `mandel` routine for 8 processors on the Cray T3D

Rather than port the rest of the Cowichan problems to the simulator and repeat the same predictions as the previous chapter, a non-deterministic example was constructed.

6.5.2 Non-deterministic example

Some parallel programs employ dynamic load balancing techniques to attempt to keep all processors busy and obtain the maximum speedup. However, the efficacy of such techniques is strongly dependent on the overheads introduced, and it may be better in practice to use a static data distribution and suffer a load imbalance than to incur these overheads.

Dynamic load balancing is the situation where reverse profiling yields little useful information and it requires a simulator to predict the behaviour.

The standard dynamic load balancing technique is the *task farm* where "worker" processes are allocated packets of work by a "farmer" process and feed their results to a "sink" process. The technique becomes unstuck if the workers are not kept busy, so it is crucial to know the order of magnitude of the overheads.

A generic task farm was constructed in MPI, with the workers taking randomly distributed times to complete tasks. This was run on the simulator with the communications models of the Cray T3D and the network of workstations, and the predicted timing diagrams are shown in figures 6.16 and 6.18. The measurements are in figures 6.17 and 6.19. The predictions and measurements for the Cray yield the same information; the communications are taking a significant proportion (between 15% and 30%) of each worker's time, and this is worsened by contention at the farmer (process 0), especially at the start of the algorithm when all workers are demanding packets from the farmer at approximately the same time.

The predictions and measurements for the network of workstations indicate that the problem is totally dominated by communications overheads which are over an order of magnitude greater than the packet computation time.

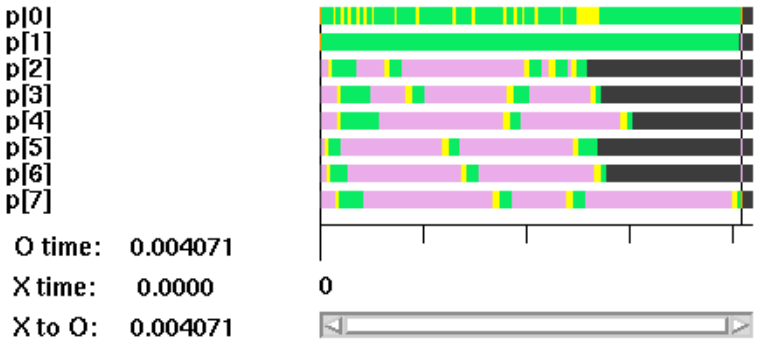

Figure 6.16: Predicted timing diagram of non-deterministic application for Cray T3D.

6.6 Conclusion

An environment for developing MPI programs on a simulator has been presented. The level of detail of the simulation model may be varied between detailed hardware models and simple equations of the MPI routine performance derived from a benchmark routine.

The SIM++ implementation of the MPI functions runs faster than the standard workstation version. This result is a good counter to the argument that

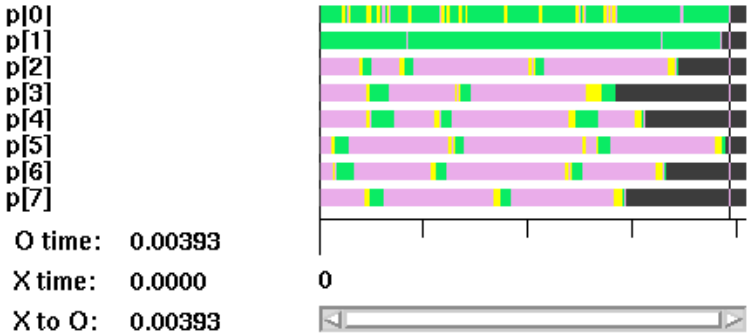

Figure 6.17: Measured timing diagram of non-deterministic application on Cray T3D.

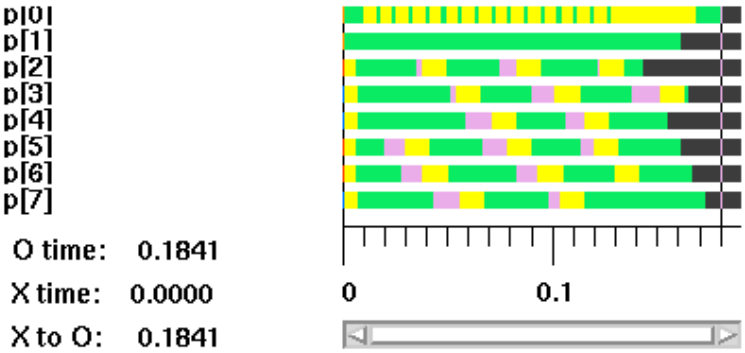

Figure 6.18: Predicted timing diagram of non-deterministic application for network of workstations.

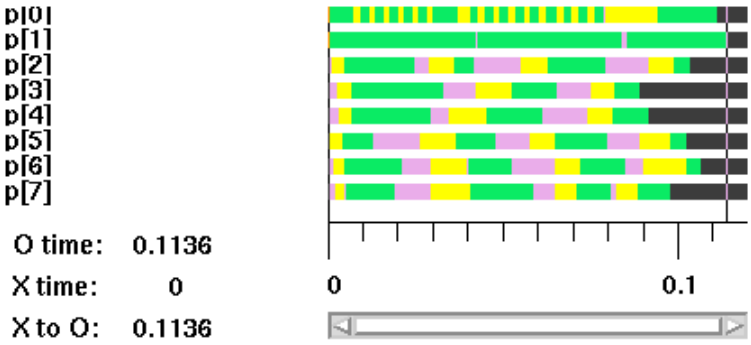

Figure 6.19: Measured timing diagram of non-deterministic application on network of workstations.

simulation is too slow to use. However, it is a sequential simulator so the run time grows with the *total* amount of computing to be done across all simulated processors.

The problems with using a simulator for program development are practical rather than theoretical. The main problems are maintaining a separate implementation of MPI (written in terms of the simulation primitives), and the fact that it requires slightly more effort to run code on the simulator than to run it on a standard MPI development platform. This means that for all but complex non-deterministic problems the **reverse profiling** technique is more appropriate.

Bibliography

[1] R. Aversa, A. Mazzeo, N. Mazzocca, and U. Villano. The PS Project: development of a simulator of PVM applications for Heterogeneous and Network Computing. In Innes Jelly and Ian Gorton, editors, *Software Engineering for Parallel and Distributed Systems : Proceedings of the First IFIP TC10 International Workshop on Parallel and Distributed Software Engineering*. IFIP, Chapman and Hall, March 1996.

[2] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. PROTEUS: A high performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[3] E.A. Brewer and W.E. Weihl. Developing parallel applications using high-performance simulation. In *Proceedings of 1993 Workshop on Parallel and Distributed Debugging*. San Diego, CA, 1993.

[4] P.M. Dickens, P. Heidelberger, and D.M. Nicol. A distributed memory LAPSE: Parallel simulation of message-passing programs. Technical Report NAS1-19480, NASA Langley Research Center, Hampton, VA 23681, December 1993.

[5] F.Brooks. *The mythical man-month*. Addison-Wesley, 1975.

[6] Message Passing Interface Forum. MPI: A Message Passing Interface. Technical report, University of Tennessee, June 1995.

[7] J. Hillston. A tool to enhance model exploitation. Technical Report CSR-20-92, Dept. of Computer Science, University of Edinburgh, 1992.

[8] F.W. Howell and R.N. Ibbett. *STATE-OF-THE-ART IN PERFORMANCE MODELLING AND SIMULATION Modelling and Simulation of Advanced Computer Systems: Techniques, Tools and Tutorials, edited by Kallol Bagchi*, chapter 1:Hierarchical Architecture Simulation Environment, pages 1–18. Gordon and Breach, 1996.

[9] F.W. Howell, R. Williams, and R.N. Ibbett. Hierarchical Architecture Design and Simulation Environment. In *MASCOTS '94: Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, January 1994.

[10] R.N. Ibbett, P.E. Heywood, and F.W. Howell. HASE: A Flexible Toolset for Computer Architects. *The Computer Journal*, 38(10):755–764, 1995.

[11] I.E. Jelly and I. Gorton. The PARSE project. In Innes Jelly and Ian Gorton, editors, *Software Engineering for Parallel and Distributed Systems : Proceedings of the First IFIP TC10 International Workshop on Parallel and Distributed Software Engineering*, pages 271–276. IFIP, Chapman and Hall, March 1996.

[12] S.K. Reinhardt and M.D. Hill et al. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proc. 1993 ACM SIGMETRICS Conference*, May 1993.

[13] *SIM++ v3.8 Reference Manual*, 1991.

[14] Southampton Novel Architecture Research Centre. *The GENESIS Distributed Memory Benchmark Suite 2.2*, 1993.