Approaches to parallel performance prediction

Fred Howell

Doctor of Philosophy
University of Edinburgh
1996

Abstract

Designing parallel programs is both interesting and difficult. The reason for using a parallel machine is to obtain better performance, but the programmer will have little idea of the performance of a program at design time, and will only find out by actually running it. Design decisions have to be be made by guesswork alone.

This thesis explores an alternative by providing data sheets describing the performance of parallel building blocks, and then seeing how they may be used in practice.

The simplest way of using the data sheets is based on a graphing and equation plotting tool. More detailed design information is available from a "reverse" profiling technique which adapts standard profiling to generate predictions rather than measurements. The ultimate method for prediction is based on discrete event simulation, which allows modelling of all programs but is the most complex to use.

The methods are compared, and their suitability for different design problems is discussed.

Acknowledgements

Thanks to my supervisor Roland Ibbett for his enthusiasm; to Professor Aspinall of UMIST for asking the awkward question which sparked this work. Thanks to Gordon Smith and Lyndon Clarke of the Edinburgh Parallel Computing Centre for providing CHIMP and MPI source code for the simulator. Particular thanks to my office mates Marcus Marr and Cristina Boeres for providing coffee.

Table of Contents

Chapter 1

Introduction

This thesis documents usable techniques for designing parallel programs with *a priori* knowledge of their run time. It asks whether the guesswork can be taken out of the design process and replaced with engineering decisions based on firm data.

There is currently a gulf between the sophisticated performance analysis techniques developed by the academic community and the techniques which are used in practice.

Because of this, the approach taken in this thesis deliberately focusses on the low level parallel programming tools actually used, in the hope that it will have an immediate practical benefit to developers. Higher level techniques can build on this foundation later.

The contribution of this work to the subject is both in the form of useful tools for characterising and predicting performance and in showing that post-mortem techniques may be supplemented by ante-natal design.

This introduction reviews performance analysis techniques described in the literature and outlines the approaches investigated in the thesis.

1.1 Parallel program design techniques

The two extremes in the art of performance analysis are PRAM style complexity theory and post mortem tracing.

In practice, programmers usually concentrate on writing programs with a clear structure and worry about the performance afterwards. This is not because they don't care about the performance, but because it takes too long to work it out.

Performance is not the only design aim of a parallel system, and may not even be the the most important. However it is the one which differentiaties parallel from sequential software development.

Various techniques for performance analysis are outlined below, including computation models, high level models, mathematical models, software and hardware engineering, real time systems development, simulation, micro-benchmarking and sequential techniques.

1.1.1 The null method

This is the common approach for developing parallel programs. If a performance analysis is done, it is an afterthought.

1.1.2 Post-hoc analysis

Many papers have been written about the performance of a program on an architecture. A good example is Singh and Hennessey's paper on an ocean modelling program [33]. These specific examples are interesting, but shed little light on how one is supposed to go about developing a different program on another architecture.

1.1.3 Higher level techniques

One approach is to restrict programs to using high level operations which have been implemented efficiently (e.g. algorithmic skeletons [9] and the parallel utilities library at the Edinburgh Parallel Computing Centre (EPCC) [8]). This has considerable appeal as it frees programmers from such low level concerns as performance. However the techniques are not sufficiently well advanced to be applied to all problems and are still an area of active research.

1.1.4 PRAM

Parallel algorithm researchers are concerned with predicting the asymptotic complexity of algorithms, where the quality of an algorithm may be expressed in big O notation (e.g. an O(log N) algorithm is "better" than an $O(N^2)$ one). This approach provides a clean, simple method of comparing algorithms, but has several major drawbacks.

The first is that the computational model is idealised and will therefore not (necessarily) have much relevance to actual implementations of algorithms. Work in progress to make the models more complex and realistic (such as the HPRAMs, other PRAMS) tends to make the model harder to use and hence less useful. The other approach, redesigning parallel computers to implement the models more effectively is fairly revolutionary and hence not likely to happen unless there

is overwhelming evidence in favour of programs being easier to design using a PRAM-based model. Interestingly, PRAM-advocates insist that a major deficiency of implicit parallelism through (say) dataflow or functional languages is that performance prediction becomes trickier.

The second drawback is that the basis for comparison (asymptotic complexity) is only valid for an infinite data size or number of processors. In practice, the constant factors may be more important for machine/program sizes of interest.

1.1.5 BSP

An interesting approach, proposed by Valiant [60] provides a computational model consisting of a sequence of parallel supersteps within which local computation is performed and communication requests are posted. Between each superstep is a global barrier operation after which all posted requests are guaranteed to complete.

Restricting synchronisation to global barrier operations (and the programming style to SPMD) simplifies the general performance prediction problem to one of estimating the maximum superstep computation time and the time for the global reorganisation of data at each superstep. The performance of a machine is characterised by three values determined experimentally: **s** is the speed of computation of a process in flops, **l** is the synchronisation latency cost in units of **s** and **g** is the number of flops per word required for all processors to communicate a message simultaneously. Hill, Crumpton and Burgess [22] present interesting results using an implementation of BSP (BSPlib) on an IBM SP/2 and ethernet, comparing simplistic pencil and paper modelling with results from a profiling version of the library.

1.1.6 LogP

An attempt to create a more realistic model based on actual machine parameters rather than an abstract ideal is LogP [10]. The parameters are **L**, an upper bound on the latency suffered by a word sent from one module to another, **o**, the overhead during which a processor is occupied sending or receiving a message, **g**, the minimum time interval between successive message transmissions of receptions, and **P** the number of processors [10]. The authors note that "Such a model must strike a balance between detail and simplicity in order to reveal important bottlenecks without making analysis of interesting problems intractable."

The parameters can be estimated using a simple benchmark routine. They provide a simple pipeline model for point to point communications. Costs for

collective communications may be expressed in terms of the point to point costs, but this is not incorporated directly in the model.

Another modelling notation is $R_\infty/n^{1/2}$ developed by Hockney and Jesshope, originally for vector performance. R_∞ is the maximum rate of transfer (for infinite message sizes) and $n^{1/2}$ is the message size which achieves half of this rate. This has some advantages over

$$startup\_time + message\_size * time\_per\_byte$$

in indicating the "break-even" point for message sizes in a usable way. Numrich [52] gives these values for point-point communication on the Cray T3D network.

1.1.7 Scalability analysis tools

The NASA AIMS/MK toolset [44] extracts a program execution graph from a run of a program and uses this to feed into a discrete event simulator. Sarukkai/Mehra offer *abstract interpretation* techniques for generating complexity estimates [55]. Dunlop et al [1] looked at estimating the workload on the floating point unit and the different parts of the memory hierarchy given Fortran source code, and used this estimate for predictions.

Another top level approach described by Driscoll [12] looks at the total time spent in communication and computation throughout the program, using a variant of Amdahl's law to predict speedups. Gustafson [18] looked at the case of problem size scaling with machine sizes.

1.1.8 Queueing model techniques

Queueing theory is a well developed mathematical technique for analysing steady state performance of queueing networks. King [36] describes the application of queueing theory to computer systems. Analytical solutions exist for simple networks but more realistic networks must be simulated. The basic parameters of a queueing model (arrival rate, queue sizes etc.) may correspond directly to design parameters.

Queueing models have been used for prediction. Liang and Tripathi [37] used a simple queueing model to analyse fork/join program graphs. Mak and Lungstrom [39] developed queueing models of architecture and program for their predictions.

1.1.9 Petri-net techniques

Petri nets [47] have been used for modelling behavioural aspects of concurrent systems, particularly detecting the presence of deadlocks. Standard Petri nets do not incorporate the notion of time, so timed and stochastic extensions are typically used for modelling performance of systems. The problem with these more complex varieties of Petri nets is that they make mathematical analysis more difficult, and the state space becomes too large to search. Even with standard Petri nets, the graphical models rapidly become incomprehensible. Hartleb [19] looked at stochastic graph techniques for parallel program performance. Graph nodes were deterministic, or used a random distribution. Reduction techniques were used for simplifying models. Wabnig [30] derived a Petri net model of the communications network from first principles and used this for performance studies.

1.1.10 Process algebras

Process algebras (CCS [41], CSP [25]) incorporate better support for modelling hierarchy than Petri nets, and are amenable to analysis using state space searching techniques.

Timed variants such as TCCS [42] may be used for proving properties such as "state X occurs *before* state Y", but are not so concerned with actual run times. PEPA [24] allows for stochastic state transitions, with standard techniques used to analyse the resultant Markov chains.

If delays are deterministic rather than probabilistic, then process algebras give no more insight than simulation.

1.1.11 Parallel software engineering

Traditional software engineering techniques (Yourdon, Mellor, etc) generate a large amount of concurrency in the initial "structured analysis" phase, which they subsequently remove to produce a sequential structured design. They have nothing to say about the problems of a parallel implementation.

Attempts to develop software engineering techniques for parallel systems such as PARSE [32] have focussed on extending dataflow techniques and defining their semantics more rigorously, but have no advice on how to build in efficiency. They are also geared towards distributed systems (a few distinct processes communicating) rather than parallel systems (many identical processes communicating).

1.1.12 Hardware engineering

Concurrency comes naturally to hardware engineers as electronic components all run in parallel. Timing issues are often central to the design, so predicted timing diagrams are drawn up early in the design.

Design tools are used to a far greater extent than in the software community, with graphical tools such as schematic editors, language based tools such as VHDL and Verilog simulators, state machine designers etc etc.

Unfortunately hardware is not software so hardware design techniques cannot be directly applied to engineering of parallel software.

The important differences are :

- software components are never specified as rigorously as hardware components.
- interactions between software components are less restricted.
- a software component may be orders of magnitude more complex than a hardware component.

Software gives the engineer infinite rope to play with, whereas hardware is always bounded by physical constraints like pin count and chip area, so by necessity hardware components are better defined than software ones. Attempting to restrict software to use a controlled interface is one of the aims of software engineering, but the temptation is always there to bypass the restrictions and use a "quick and dirty" technique. Doing this in a hardware design is of course also possible, but much less common. Having to cast ideas into the stone of a circuit board encourages cleaner designs than does the free form of software.

1.1.13 Real time systems

The area of real time systems covers similar timing issues to that of parallel programming, but a poor design may be fatal rather than just inefficient. The emphasis is on predictability rather than on absolute performance; an implementation must guarantee that a deadline is met.

The Flex language [35] inserts `#pragma` comments with the expected timing equation for each section of code. The MAXT approach [49] attempts to calculate the maximum execution time of programs using software annotations and an extra compilation step. The restrictions of this method are that compile time predictions of loop counts are not always possible and also recursion is not handled.

Park and Shaw [46] present a timing schema approach which benchmarks the performance of a generalised assembler code on an architecture (with instructions such as `mov`, `add`, `mul` etc.) and then parses high level source code in terms of these instructions.

1.1.14 Direct execution simulation

Direct execution simulators allow detailed modelling of hardware and tweaking of all kinds of parameters. Indeed Brewer [5] recommends using simulation as a development platform in preference to running on a machine, based on experience with the Proteus simulator [4] of shared memory software on the CM-5 machine. The network parameters are fed in using a network model such as that described in [2] or by doing a detailed hop by hop simulation.

One possible criticism of this approach for software development is that the models take too long to construct and verify, and it is no easier than running on the actual machine. Since the models are hidden from the programmer (behind the mystique of the simulator), it is another post-mortem like approach, the only difference being that the actual machine is not used.

The advantage of this detailed approach is that network contention and other such issues may be modelled as accurately as desired.

An example of simulation applied to message passing software is PS [3], a direct execution simulator for PVM based on the Ptolemy simulation system [6]. It uses a complex model of an ethernet interface for its communications subsystem, and has been used for PVM applications running on a small number of workstations across a network. Pouzet [48] described a simulation tool for Transputer applications.

Fahringer [13] developed a system for guiding compiler optimisation as part of the Vienna Fortran Compilation System. The system statically computes a small set of parameters which characterise the overall behaviour of a parallel Fortran application.

1.1.15 Benchmaps

At the boundary between simulation and analytical techniques less work has been done. *Benchmaps* were developed by Toledo [59, 58] for prediction of data parallel programs. His technique relies on benchmarking the operations of a data parallel language (NESL), fitting a linear equation to the data, and applying the model to a running program. The memory hierarchy is modelled in a simple way with different cost models applying depending on whether or not the data is likely to fit

inside the cache. He reported errors of about 33% in predictions of performance on Sun workstations and the CM-5. Cache conflicts in the SGI Indigo workstation meant that his method would only predict performance to within a factor of about 15. Saavedra developed a micro benchmark approach to characterise the low level performance of the KSR1 memory system [53].

1.1.16 Post mortem

If performance analysis is done at all by programmers at the moment, they are most likely to use one of the post mortem trace analysis tools. Examples include Paragraph [20], Pablo [50], Vispad [26], the Upshot tool included with MPICH [43] and XMTV included with the LAM implementation of MPI [7].

They work by instrumenting a program with tracing commands, and generating a trace file with time stamps.

Post mortem techniques measure one run of the performance on one machine (and say nothing about the performance on other machines, or with different data sizes).

1.1.17 Sequential code analysis techniques

Tools exist for optimisting code on sequential machines, such as `prof`, `gprof`, the `SPARCworks analyzer` etc. These measure figures such as the number of times each function is called and the percentage of time spent in each. MacDonald [38] describes methods for analytical predictions of sequential code execution times.

1.2 Thesis overview

The methods described in this thesis are based on the standard message passing model MPI [15]. Message passing was selected for two reasons; it is more standardised than shared memory and its explicit parallel nature cries out for a design technique.

The performance of the interface is characterised by a routine which generates datasheets for each MPI function. This characterisation of the primitives is performed in the same spirit as Toledo's benchmapping approach for data parallel programs [59]. The characterisation takes the form of an equation for each MPI function, along with graphs displaying the data from which the equation was derived. Chapter 3 describes the routines used to characterise the performance of parallel building blocks and generate the data sheets. (figure 1.1).

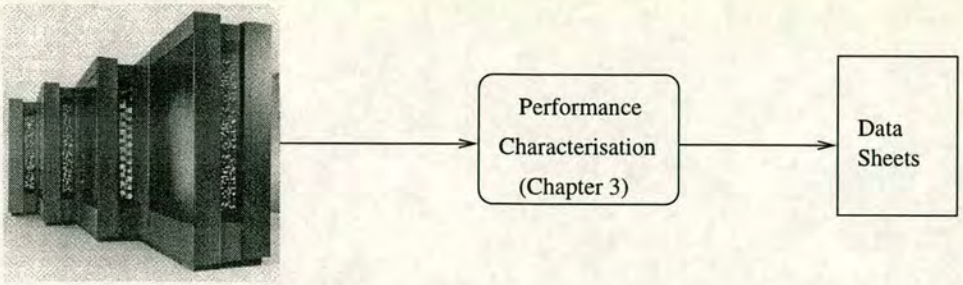


Figure 1.1: Performance predictions are based on automatically generated data-sheets for an architecture.

These data sheets may be used as they are for initial design. A simple calculating utility for evaluating the equations for given parameters was written to help with this.

Pencil and paper analysis becomes tedious and time consuming for all but the simplest program, so three computer aided techniques with increasing levels of sophistication were developed to help use the data sheets for practical development. Figure 1.2 gives an overview.

The first technique uses the data sheet results with a graphing package for rapid evaluations of the scalability of programs. This approach is presented in chapter 4. This allows experimentation at an early stage into the top level behaviour of algorithms.

A finer grain approach is presented in chapter 5. This uses the standard profiling mechanism of MPI to insert timings evaluated from the data sheets, a technique which is as easy to use as normal profiling. This allows automatic calculation of the expected timing diagrams, and copes with data dependent timings, something which compile time analysis techniques cannot handle. The results are compared with timing diagrams produced from standard profiles to assess the accuracy which can be expected from the approach. A similar approach applied to the simpler BSP model was described by Hill et al [22].

Chapter 6 investigates a simulation tool which extends the approach above to handle non deterministic programs as well as deterministic ones. It also permits inclusion of detailed hardware models in addition to the data sheet models.

Chapter 2 describes the experimental techniques used to evaluate the various approaches. A suite of problems requiring a variety of parallel implementation strategies was chosen to test the ease of use and accuracy of the design techniques.

These strands are drawn together in the conclusion (chapter 7) which evaluates the techniques and presents plans for future development.

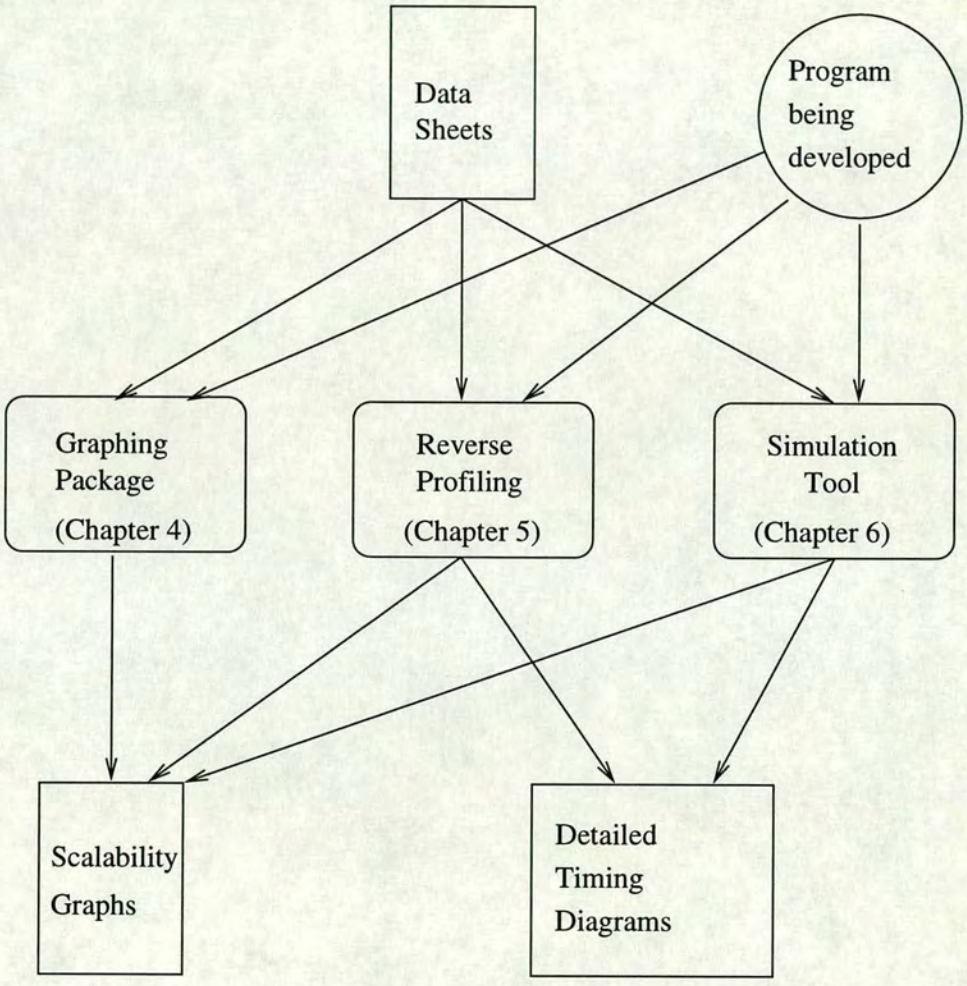


Figure 1.2: Three techniques for using the datasheets to help design programs.

Chapter 2

Experimental Approach

To evaluate the efficacy of the design techniques, a suite of parallel problems was chosen and coded in MPI. The suite of problems is described in section 2.1. Other MPI benchmarks are now available, organised by the PARKBENCH committee [21].

In his paper on performance prediction of data parallel programs, Toledo [59] stated that:

> *"We believe that the most important open question in performance prediction today is how to assess and verify the accuracy of performance models. Without the means to assure the accuracy of models it is difficult to put them in production use."*

Section 2.3 describes the profiling technique used to extract actual timings from runs on parallel machines. Section 2.4 describes how detailed comparisons of predicted and actual trace files are performed. Section 2.5 describes the techniques used for comparing actual and predicted scalability.

Assessing how easy the design techniques are to use is more subjective than assessing accuracy. A subjective comparison based on experiences using the design techniques is therefore given in the final chapter.

2.1 The test suite: the Cowichan problems

The Cowichan problems [1] were set by Wilson [61] to assess the usability of parallel programming systems. The suite consists of fourteen problems intended to be implemented in parallel. Some of the problems such as the vector difference routine `vecdiff` are simple to code for a parallel machine. Others such as invasion percolation `invperc` present more difficulties.

[1] "Cowichan" is a place name on the NW coast of N. America

The problems were set to provide an objective basis for comparing how easy different parallel programming systems are to use; the intention was that different groups would code the problems using their preferred tools (shared memory, threads, HPF, MPI etc.) and record the time required to write the parallel versions and the problems faced. A comparison of the problems encountered in porting the problems would enable the usability of MPI, threads, shared memory, HPF etc. to be compared.

A sequential version of all the problems was coded by Wilson in ANSI C, and this was used as the reference implementation for checking that the parallel versions produced the correct results.

The MPI version of the problems was written by Howell and Marr using C++.

The Cowichan problems are specified fully in [61], but brief descriptions of the problems are included below, along with notes on the MPI implementation. Appendix A includes a brief summary of MPI.

2.1.1 `mandel`: Mandelbrot Set

This module computes the mandelbrot set as a matrix of integers (figure 2.1). Each point of the set may be computed independently so it is a simple routine to parallelise. In the MPI implementation, each process is allocated an equal slice of the set to work on.

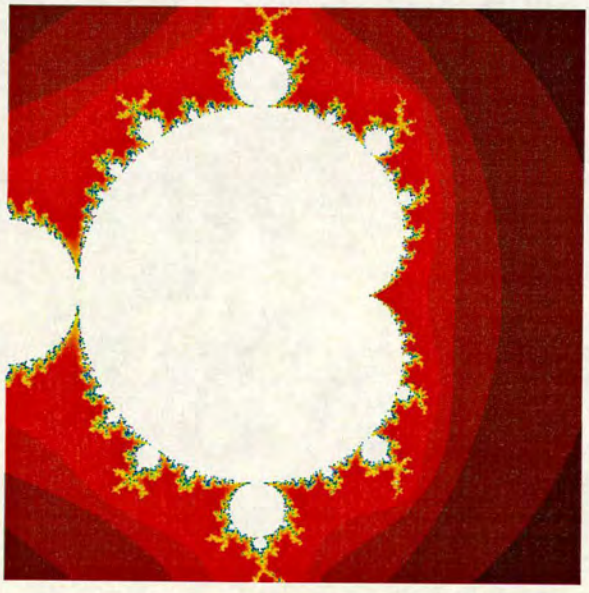


Figure 2.1: The output from `mandel`.

2.1.2 `randmat`: Random matrix generation

This module generates a matrix of pseudo random integers, with a given random number seed (figure 2.2). The aspect of this problem which complicates the parallel implementation is that each successive point in the matrix is computed from the previous one;

$$r_{i+1} = (r_i * a + b) \text{modulo} 2^{32}$$

i.e. there is a sequential dependency. The output must be repeatable and independent of the number of processes; starting each process with the same seed leads to a distinctly non-random striped effect.

The parallel version solves this problem by computing the initial seeds for all processes using an order log(N) algorithm. After this, all may continue independently to compute their area of the matrix.

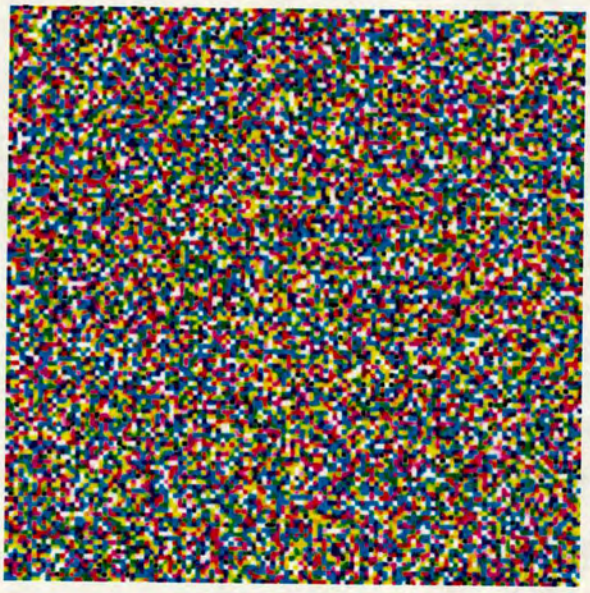


Figure 2.2: The output from `randmat`.

2.1.3 `half`: Two-dimensional shuffle

This module shuffles the values of a matrix along both the rows and the columns. Figure 2.3 shows the algorithm applied to the Mandelbrot set. It exercises the communications facilities of MPI.

2.1.4 `life`: The game of life

This module simulates the evolution of Conway's game of life, a 2D cellular automaton. Figure 2.4 shows the algorithm applied to boolean matrix generated by

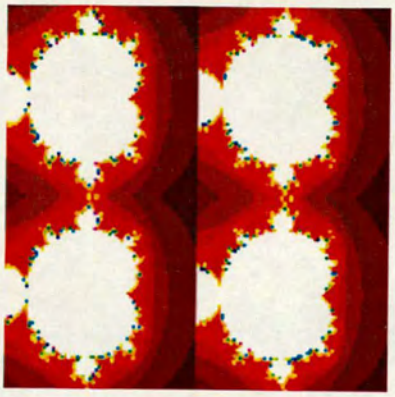


Figure 2.3: The output from `half`.

thresholding a random matrix. The routine uses nearest neighbour communications to calculate the number of neighbours each point has. Global synchronisation is also necessary to keep all processes in step.

Figure 2.4: The output from `life`.

2.1.5 `thresh`: Histogram thresholding

This module performs histogram thresholding on an image. It constructs a binary image from all the pixels of an integer image with an intensity in the top $p\%$ of all pixels. This requires a global reduction to find the highest valued pixel, followed by a local histogram computation. The histograms are then merged (by summing across all processors), the threshold is computed and applied to the image in parallel. Figure 2.5 shows the results of thresh applied to a random matrix (the output of 2.1.2).

2.1.6 `outer`: Outer product

This module takes a vector of point coordinates and forms a dense symmetric diagonally dominant matrix of the distance of each point from every other point,

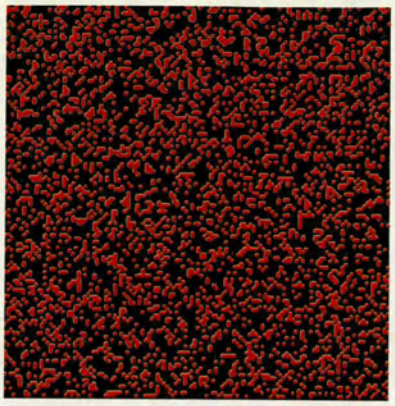


Figure 2.5: The output from histogram thresholding a random matrix.

and a vector of the distance from each point to the origin. Figure 2.6 shows a sample output matrix.

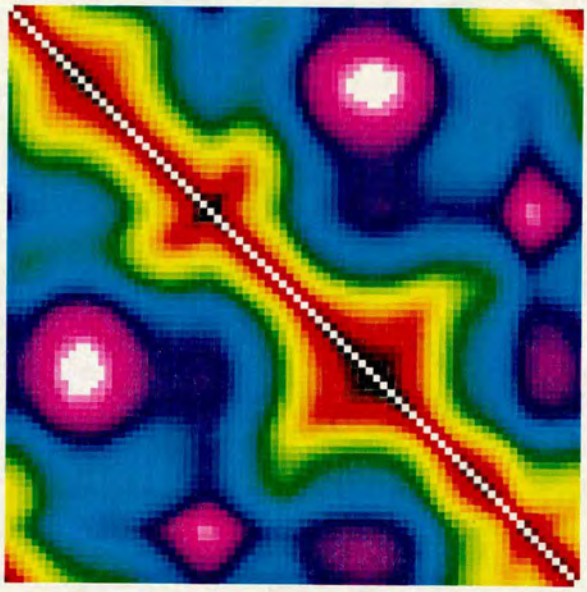


Figure 2.6: The output from the outer product module.

The vector generation is simple; each process has a copy of all points and works on its local section of the distributed vector. The matrix generation is nastier. The simplest solution has each process computing the sections of both the upper and lower triangles with a global reduction (implemented with `MPI_Allreduce`) to get the diagonal. This does twice as much calculation as necessary (as the upper triangle is a mirror copy of the lower), so an alternative would be to load balance one of the triangles then copy that triangle onto the other. Both were implemented for comparison.

The decision of whether to load balance and copy the triangles or to do twice the computation requires a performance specification of the MPI routines. The

triangle copy is difficult to implement in MPI, requiring complex data types and a new operation: `MPI_Alltoallvi`.

2.1.7 `elastic`: Elastic net simulation

This module solves the travelling salesman problem using the elastic net algorithm. The algorithm deforms an elastic circular loop towards the city locations, using an algorithm for the "force" exerted on the elastic by each city and for the elastic force keeping the loop together. Multiple iterations are run, deforming the loop until all cities are connected. Figure 2.7 shows the first iterations of the algorithm, with the circular ring being stretched towards the cities.

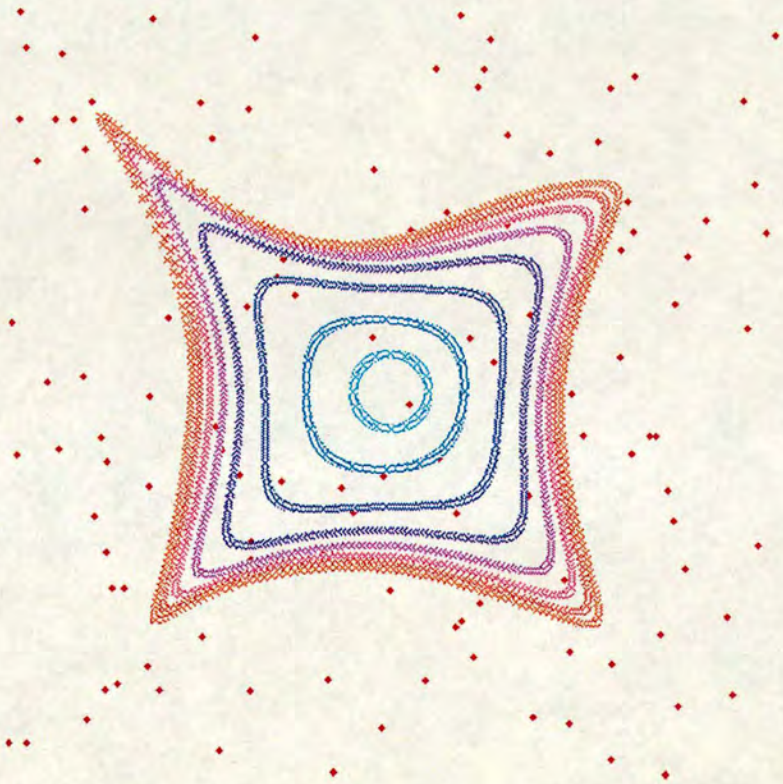


Figure 2.7: In the first steps of the elastic net simulation a circular ring is stretched towards the cities.

The vector of city locations is fixed so it is copied to all processes. The vector of points on the loop is distributed evenly, with neighbour communications. In MPI this boundary communications step is not trivial as there is a special case when fewer than three points of the elastic loop are stored on a process. The boundary exchange was implemented as the sequence `send, send, recv, recv` which may deadlock given limited buffer space but is simple. This is reasonable as the buffer space requirement is only for four real numbers. The alternative

would be for the odd numbered processes to send and the even numbered ones to receive, then vice-versa.

2.1.8 `invperc`: Invasion percolation

Invasion percolation simulates the displacement of one fluid by another in fractured rock. The input is a matrix of integers representing rock densities. In each iteration, all neighbours of all filled cells are examined, and the one with least resistance (i.e. lowest density) is filled. The output is a fractal shape such as that shown in figure 2.8, where the white dots show the filled locations.

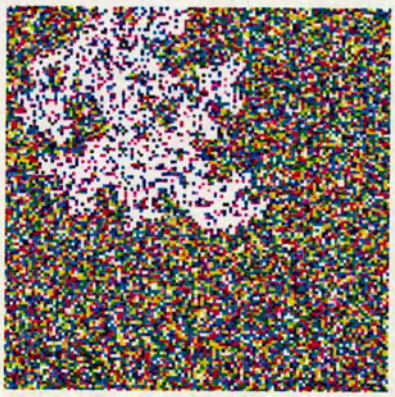


Figure 2.8: The output from invasion percolation run on the random matrix produced by `randmat`.

This is an awkward problem to parallelise. The initial implementation is *not expected to give a speedup.* It uses a distributed queue, where each process stores elements of the queue which lie on the local slice of the matrix. The `enqueue` and `dequeue` operations are called by all processes but operate on only one at a time. The algorithm is inherently sequential and the only possible speedup would be from the shorter queue insert times.

2.1.9 `product`: Vector/matrix product

This module performs the product of a real matrix with a real vector, returning a real vector. The parallel version distributes the rows of the matrix and copies the entire vector to all processes. Each process then computes its local section of the results vector.

2.1.10 `sor`: Successive over-relaxation

This module solves a system of linear equations using the successive over-relaxation iterative technique. The parallel version evenly distributes the rows of the mat-

rix and result vector. At each iteration step each element of the result vector is "relaxed" towards the correct solution by applying a factor computed from the error in the current value. Each process then obtains a local copy of the entire updated result vector for the next iteration. This continues until the solution is within a defined tolerance, or the maximum number of iterations is reached.

2.1.11 `gauss`: Gaussian elimination

This module performs Gaussian elimination to solve a set of linear equations. The output is a real vector containing the solution. In the parallel version each process computes its local pivot. A global communication step then selects which pivot element to use.

2.1.12 `norm`: Point normalisation

The problem is to normalise a vector of point coordinates to lie in the unit square. Figure 2.9 shows example output.

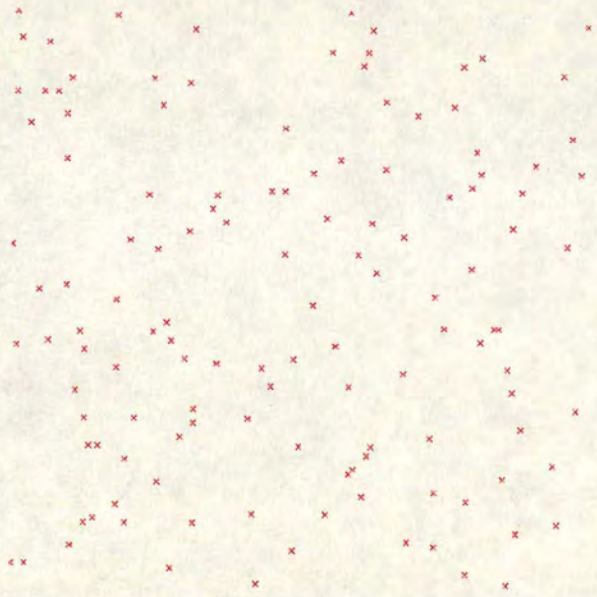


Figure 2.9: The output from `norm`.

The point vector is distributed evenly. A global reduce is required to determine extremities, followed by a scaling of the local data.

The global reduction needs to find the maximum and minimum point locations for the scaling. There is a choice of four `MPI_Allreduces` (minx, maxx, miny, maxy), or a single `MPI_Allreduce` with a user defined reduction operation. The former was selected for simplicity although the latter is likely to be faster.

2.1.13 `winnow`: Weighted point selection

This module converts a matrix of integer values into a vector of points. A boolean mask matrix is used to select values from the integer matrix. These values are sorted, and the row/column coordinate of every *Nth* is added to a point vector which is returned. The sequential implementation is straightforward, using loop indices to select particular items of data. With distributed data structures in MPI however it is extremely awkward, requiring extensive use of user constructed datatypes to perform the strided redistribution of data. Each process sends and receives different amounts and types of data to/from all the others.

2.1.14 `vecdiff`: Vector difference

This module returns the maximum difference between corresponding elements of two real vectors. Local differences are computed first, and a global operation returns the overall maximum.

2.2 Techniques used in the MPI implementation

The following notes describe the fundamental parallel data structures and I/O techniques used in the MPI implementation of the Cowichan routines.

2.2.1 Distributed data structures

Matrices and vectors are often distributed evenly across the processes to balance the workload. The standard method of distributing and gathering data in MPI uses the collective communications functions. The example below shows how a matrix distributed across processes may be gathered so that each process has a copy of the entire matrix.

```
int local_matrix[localnrows][ncols];
int global_matrix[nrows][ncols];

int  *counts = /* number of elements on each process */;
int  *displs = /* global offset of 1st element on each process */;

MPI_Allgatherv(
  local_matrix, localnrows*ncols, MPI_INT,
  global_matrix, counts, displs, MPI_INT,
  MPI_COMM_WORLD );
```

To allow such redistribution of data, each process must maintain the arrays `counts` and `displs` to store the number of elements on each process and the displacement from the start of an array. The arrays are combined into a class to simplify use of collective operations:–

```
class distribution {
  int *counts;
  int *displs;
public:
  // Methods to generate useful distributions.
}
```

The inheritance facility of C++ allows a "distributed vector" or "distributed matrix" to be defined by deriving from both `class distribution` and `class vector<Type>`:–

```
class vector_d<Type> : public vector_t<Type>, distribution {
public:
  // Extra methods peculiar to distributed vectors
}
```

Extra methods can then be added to allow global access to distributed data structures, hiding the local offset calculations. These distributed structures may be passed as function arguments and the `counts` and `displs` arrays are available for calling the MPI collective routines.

2.2.2 File I/O

Parallel file I/O is not a standard part of MPI, although all parallel programs will require some I/O. The solution adopted for the Cowichan problems was to perform standard I/O from the root process alone, and to scatter or gather the data to all processes as appropriate. This leads to a heavy I/O cost for each module, as there is a sequential phase before and after the computation which is not shortened as more processes are added. This I/O phase was quantified for the performance evaluations, but the main focus of comparisons was the parallel sections of code, since it is anticipated that truly parallel I/O routines will become the norm eventually.

2.2.3 Graphics

An X windows graphics display was written using the facilities of the MPI implementation on workstations (LAM). The facilities are primitive, offering basic

pixel and area routines but suffice for the production of some interesting pictures such as those illustrating this chapter.

2.2.4 Problems with using MPI

The most painful aspect of using MPI is the datatype definition. The facilities provided for constructing user defined datatypes are powerful but awkward to use. To define a simple `struct` of a `float` and an `int` takes about 10 lines of code and provides plenty of scope for mistakes.

The least pleasant aspect of message passing is the extra code required for computing offsets into distributed data structures. More code is also needed to handle special cases such as how a neighbour exchange should work with less than three elements on a process.

2.3 Measuring performance

To compare predicted with actual timings, a reliable method for obtaining the actual timings was needed. The approach involved a mixture of automatic profiling (for the times of the MPI functions) and user profiling (for the times of the different application phases).

The automatic profiling was accomplished using the MPI profiling library. The user profiling was done using some simple macros:

```
time_set(MPI_Wtime());
// Input
time_mark(''input'');
// Broadcast arguments
time_mark(''arg broadcast'');
// Compute results
time_mark(''compute'');
// Write output
time_mark(''output'');
time_total(MPI_Wtime());
time_trace(''test'');
```

These produce a trace file for each run. A separate tool was written to perform successive runs with a range of data sizes and number of processes and to collect the results.

Timing results from a single run may be displayed using the timing diagram tool from the HASE simulation environment [31] described in chapter 6. This shows a zoomable timing diagram with bars for each of the process states.

2.4 Comparing single run predictions with measurements

The obvious way to check the accuracy of the prediction is to time an actual run and compare it with the predicted total time, e.g. (the numbers in the following tables are for illustration only).

Predicted time	Measured time	Ratio
1.23	2.46	2.0

It is possible that this method could indicate that a prediction is perfect whereas in fact a gross overestimate for one part of the time may be serendipitously compensated for by an underestimate for another part. To gain a deeper perspective into the accuracy thus requires looking at more detail than the total run time.

At the next level of detail, the measured/estimated times for *phases* of the application may be compared.

Phase	Predicted time	Measured time	Ratio
Load	1	3	3.0
Bcast	0.3	0.3	1.0
Compute1	3	1	0.3
Compute2	2	2.4	1.2
Gather	7	3.5	0.5
Wr Results	0.32	0.35	1.1
Total	14.6	10.2	0.7

This gives more detail on where the technique is over or underestimating the time. However, even this amount of detail is not sufficient to evaluate whether the technique could be gainfully applied to developing a new application, and it is necessary to look at a finer level of detail to check that the models of the building blocks of an application are applicable.

At this level the possibility of measurements interfering with the system emerges and the quantity of data starts to explode as the comparison is between *tracefiles* of predicted and actual execution. Each iteration of each loop is included in the trace. There is too much data to display in a table and it is difficult to

provide any meaningful comparison. Displaying both traces as timing diagrams gives a visual comparison but extracting numbers is more difficult.

To illustrate the problem, part of a sample trace file from Upshot's ALOG format [7] is given below. On each line, the first number is the event type (e.g. 1 = start broadcast); this is followed by the process number and three zeros (left for expansion). The last number on a line is the time stamp, and this is followed by the name of the event.

```
1 1 0 0 0 680407 start broadcast
2 1 0 0 0 682514 end broadcast
7 1 0 0 0 682693 Start Sync
1 3 0 0 0 682774 start broadcast
2 3 0 0 0 683227 end broadcast
7 3 0 0 0 683322 Start Sync
2 0 0 0 0 687328 end broadcast
7 0 0 0 0 687417 Start Sync
8 0 0 0 0 690727 End Sync
3 0 0 0 0 690796 start compute
8 2 0 0 0 692187 End Sync
3 2 0 0 0 692361 start compute
8 1 0 0 0 695886 End Sync
```

The problem would be somewhat simplified if the predicted and measured tracefiles differed only in the values of their timestamps, but the asynchronous nature of parallel systems means that the ordering of tracefiles often varies. All the Cowichan routines were written to use deterministic patterns of communications which enables traces to be compared using the utility described below.

2.4.1 A trace comparison utility

To address the problem of comparing predicted and measured executions, a *trace comparison utility* was written in C++. Figure 2.10 illustrates the technique.

The utility is used from the Unix command line:

```
cmptrace <infileA> <infileB> <outfile>
```

The input format is trace files in SIM++ format [56]:-

```
$types
State COMPUTE SEND RECV REDUCE BARRIER BCAST GATHER \
```

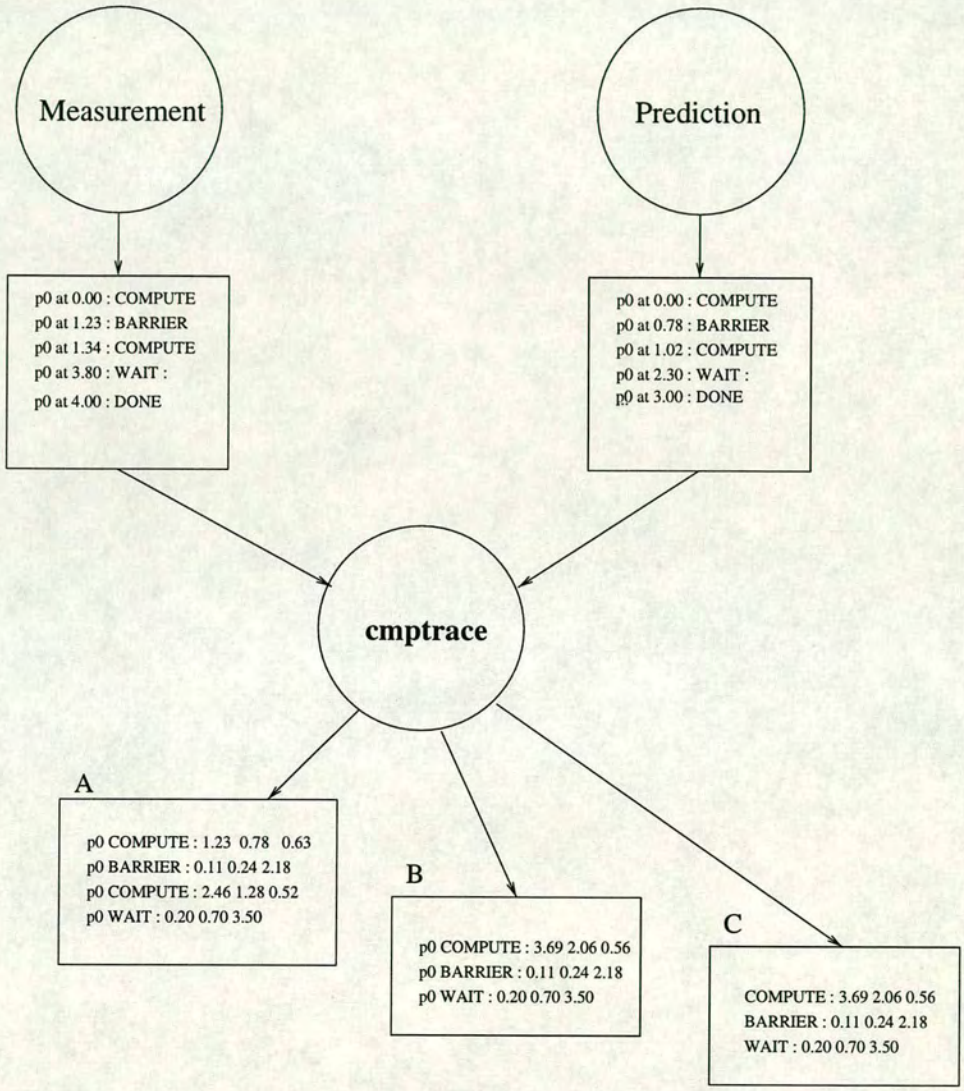


Figure 2.10: The trace comparison utility `cmptrace`.

```
        ALLGATHER ALLREDUCE COMMSPLIT DONE
$bars
p[0] State
p[1] State
$events
u:p[0] at 0.0:          P BARRIER
u:p[0] at 1.0:          P REDUCE
u:p[0] at 2.0:          P COMPUTE
u:p[0] at 3.0:          P BARRIER
u:p[0] at 4.0:          P REDUCE
u:p[0] at 5.0:          P COMPUTE

u:p[1] at 0.0:          P BARRIER
u:p[1] at 1.0:          P REDUCE
u:p[1] at 2.0:          P COMPUTE
u:p[1] at 3.0:          P BARRIER
u:p[1] at 4.0:          P REDUCE
u:p[1] at 5.0:          P COMPUTE
// ...
```

There are several output formats, with varying levels of detail. The first (**A** in the diagram) is a line by line comparison of *all* the events in the trace file. It has one line for each line in the input trace files, so may be very large.

```
p[0] BARRIER    <t1> <t2> <t2/t1> <trclineno>
p[0] REDUCE     <t1> <t2> <t2/t1> <trclineno>
p[0] COMPUTE    <t1> <t2> <t2/t1> <trclineno>
p[0] BARRIER    <t1> <t2> <t2/t1> <trclineno>
p[0] REDUCE     <t1> <t2> <t2/t1> <trclineno>
p[0] COMPUTE    <t1> <t2> <t2/t1> <trclineno>
// ...
p[1] BARRIER    <t1> <t2> <t2/t1> <trclineno>
p[1] REDUCE     <t1> <t2> <t2/t1> <trclineno>
p[1] COMPUTE    <t1> <t2> <t2/t1> <trclineno>
p[1] BARRIER    <t1> <t2> <t2/t1> <trclineno>
p[1] REDUCE     <t1> <t2> <t2/t1> <trclineno>
p[1] COMPUTE    <t1> <t2> <t2/t1> <trclineno>
// ...
```

The next output format (**B**) collates totals for each process in each state:

```
p[0] BARRIER        <t1> <t2> <t2/t1>
p[0] REDUCE         <t1> <t2> <t2/t1>
p[0] COMPUTE        <t1> <t2> <t2/t1>
p[1] BARRIER        <t1> <t2> <t2/t1>
p[1] REDUCE         <t1> <t2> <t2/t1>
p[1] COMPUTE        <t1> <t2> <t2/t1>
```

and the last summary format (**C**) gives totals for the states across all processes:

```
BARRIER    <t1> <t2> <t2/t1>
REDUCE     <t1> <t2> <t2/t1>
COMPUTE    <t1> <t2> <t2/t1>
```

The trace comparison utility uses the SIM++ trace format but could be extended to use other formats such as ALOG or Pablo.

The utility enables detailed comparisons of prediction techniques with actual measurements, and helps to pinpoint the failings (and successes) of the prediction techniques. It may also be used to compare the detailed performance of runs of the same program on different machines. Another use is determining the repeatability of measurements by comparing successive runs on the same machine.

Only one timing (predicted or measured) is given for each phase even though in a MIMD system each process will finish a phase at a different time. The time for a phase is taken to be the maximum time taken by all processors in the group.

2.5 Multiple runs

The designer of a parallel program may be designing for a fixed machine and problem size, in which case a performance prediction technique which provides a single number for the run time would suffice. However it is more likely that the design will have to encompass a range of machine and/or problem sizes leading to 2D graphs or 3D surfaces.

In addition, each sample point on the surface will be taken from a distribution (since delays will vary statistically), so the comparison must be between two 3D probability distributions.

Experiments to measure how performance varies with different data and machine sizes were controlled using an `experimentor` routine written using `Perl`.

Comparison routines were written to compare two graphs, returning a third graph giving the ratio of the first two.

Another approach would be to fit curves to both sets of data and compare the coefficients. However this would involve guessing the form of the equations, which may well be very complicated (and possibly non-linear).

The following subsections describe two utilities developed for displaying and comparing 3D surfaces obtained from multiple run experiments.

2.5.1 mkgraph: a utility to generate graphs

`mkgraph` is a utility for generating 3D surfaces. It is used from the Unix command line:

```
mkgraph <infile> <outfile>
```

The input format is:

```
# <nprocs1> <ndata1>
<phase1name> <time>
<phase2name> <time>
...
<totalname> <time>
# <nprocs2> <ndata2>
<phase1name> <time>
<phase2name> <time>
...
<totalname> <time>
... etc
```

As output it generates a separate data file for each phase which includes the speedup from the single processor time. It also produces GNUplot script files for displaying the data as a 3D surface.

2.5.2 cmpgraph: a utility to compare graphs

A utility for comparing two 3D surfaces was developed, `cmpgraph`. The utility is used from the Unix command line:

```
cmpgraph <infileA> <infileB> <outfile>
```

The format of the two inputs is the same as for `mkgraph`. The utility computes the ratio of the two input surfaces, and generates the data files and 3D GNUplot scripts for displaying them.

2.6 Conclusion

This chapter has described the suite of real MPI programs used for evaluating the design techiques described in later chapters. It is difficult to assess the accuracy of a prediction. The ultimate comparison is between predicted and measured trace files, so a tool was developed to perform this detailed line by line comparison (section 2.4.1). Scalability comparisons are also important, and tools were developed for controlling multiple runs (section 2.5) and for comparing the timing results (section 2.5.2).

Chapter 3

The Performance Characterisation of MPI Functions

It is unusual for detailed performance information to be available to a parallel programmer setting out to design a program. This is rather a serious omission, since the point of using a parallel machine is to obtain better performance, and without performance information it is hard to make design tradeoffs. This chapter presents a method for characterising an MPI implementation which produces approximate equations for the behaviour of each function. The result is an automatically generated LaTeX datasheet for the MPI implementation. A summary file suitable for computer aided performance prediction is also produced by the routines.

3.1 Introduction

A programmer sitting down to design a program for a parallel machine with (say) an MPI manual will encounter design choices with precious few design guidelines. For example, if the program makes extensive use of triangular matrices, is it worth redistributing the data to balance the computation load? Or would the cost of redistribution outweigh the benefits of balanced load and make it faster to suffer the imbalance? The answers to such questions, of course, depend on the relative costs of computation and communication which in turn depend on the architecture. Many attempts at characterising architectures have been made and abstract "bridging models" have been proposed to simplify life for programmers. Examples include PRAM (shared memory, communication is free) and LogP. As Foster notes [16], they are not especially applicable to program writing. None of the models will tell a programmer that (say) an `MPI_Barrier()` executes instant-

aneously while an `MPI_Bcast()` requires **O(N*P)** time on the chosen machine.

What would be useful in practice is a table giving the performance of each `MPI_` function as an equation with the amount of data and number of processes as parameters. Any such equation is bound to be a simplification, so information about the accuracy of such estimates should also be included. It is not obvious what form the equations should take. There is a compromise between accuracy and usability; an equation such as:

$$t = 0.15644 + (N_{procs})^{2.2345} * 1.23456 + (N_{data})^{1.0017}$$

may describe the behaviour of a function accurately but would be cumbersome to use in practice.

The best set of design rules would assign simple costs (like **1**, **N**, **0**) to operations enabling rapid paper calculations to resolve design choices like the triangular matrix dilemma above. The most familiar model following this path is the PRAM which assigns zero to the communications cost. Unfortunately such simple design rules are unlikely to describe reality so something more complex is needed. It is interesting to reflect that such informal design rules govern programmers at present, as in "I'll use an asynchronous send here since it will be 'much quicker' than a standard send"; "better not put a barrier in this loop as it will dominate the time".

Simple latency and bandwidth measurements are frequently quoted for machines, for example Nog and Kotz [45]. Ciula [34] compared latency and bandwith for workstations connected by ATM, FDDI, FCS and ALLNODE switches. These were point to point measurements between two machines:

	Latency (us)		Bandwidth (MB/s)	
Network	PVM	MPI	PVM	MPI
ATM	677	702	9.2	9.2
FDDI	913	1115	6.4	7.8
FCS	944	1253	6.4	6.9
ALLNODE	546	546	3.7	4.3

Actually making use of this information is difficult. It shows the cost of bouncing a message from point to point, but not the delay the sender undergoes before the next instruction. It also provides no clue as to the performance of collective communications or to the performance of a set of point to point operations performed concurrently (as in for example a boundary exchange).

The rest of this chapter describes MPI characterisation routines aimed at providing more concrete information to guide designers. Section 3.2 discusses

alternative ways of presenting performance information. Section 3.3 describes how the communications functions are timed. Section 3.4 discusses some of the problems inherent in timing communications on parallel machines. Section 3.5 explains how the data sheets are generated, with some sample extracts given in section 3.6; finally section 3.7 concludes.

3.2 Presenting performance information

Three ways of presenting the performance information for MPI functions were considered; interpolating directly from measurements, using a curve fitting technique, and categorising functions into a few simple buckets (such as "quick" and "slow"). A curve fitting technique was chosen as it provides a balance between simplicity and accuracy.

3.2.1 Interpolation from measurements

If enough data values were to be measured for the required architecture, then the performance for a particular function could be read from a table (or extrapolated from adjacent timings). This would be cumbersome without a computer tool to help, but could be approximated by referring to a graph of measurements.

3.2.2 Best fit equations

Curve fitting techniques exist to fit an equation to a set of measurements. The best form of the equation is not obvious; the aim is a simple and accurate equation. A simple form was tried initially, with three coefficients:

$$t = \text{c\_coeff} + N_{procs} * \text{p\_coeff} + N_{data} * \text{d\_coeff}$$

This plane didn't fit the curved surface of many of the collective routines (see figure 3.1) so an extra term pd_coeff was introduced:

$$t = \text{c\_coeff} + N_{procs} * \text{p\_coeff} + N_{data} * \text{d\_coeff} + \text{pd\_coeff} * N_{procs} * N_{data}$$

This improved many of the fits (at the cost of making the equation more complex), but was not sufficient for functions whose time grew with $log(N_{procs})$, so an extra term was introduced:

$$t = \text{c\_coeff} + N_{procs} * \text{p\_coeff} + N_{data} * \text{d\_coeff} + \text{pd\_coeff} * N_{procs} * N_{data} + \text{logp\_coeff} * \log(N_{procs})$$

This made the fits better (by eye), but some functions had a dependency on $\log(N_{procs}) * N_{data}$, so yet another term was added.

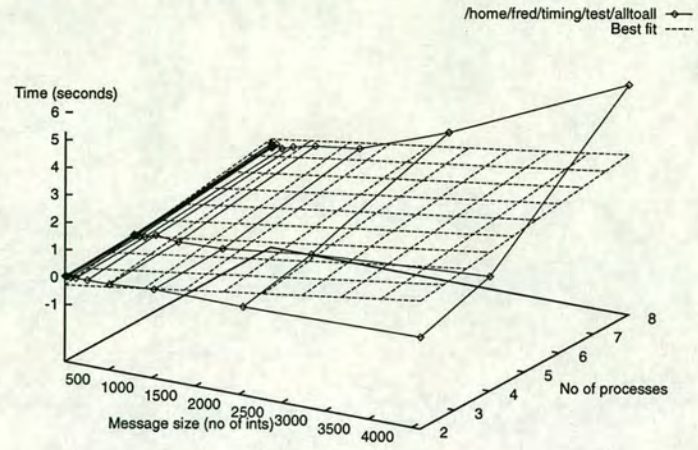


Figure 3.1: Example measured times for `MPI_Alltoall()` with fitted *plane.*

By this stage the equations gave reasonable fits in all cases measured (on Sun networks and the T3D) but the equation was too complex to allow quick estimates.

Another approach, suggested by Marr [40], was that rather than have one form of equation for all functions, it might be better to try many different, simpler, equations and see which fits best. The equations for the time of an operation in terms of the number of processes in the group p and the message size d take the form of a constant factor, a "startup parameter" dependent on the number of processors, and a "data dependent" factor dependent on the message size and the number of processors:-

$$t(p,d) = \text{c\_coeff} + \text{s\_coeff} * \mathbf{startupfn(p)} + \text{d\_coeff} * \mathbf{datafn(p,d)}$$

$$\mathbf{startupfn(p)} = \text{one of} \begin{cases} p \\ \log(p) \\ p^2 \end{cases}$$

$$\mathbf{datafn(p,d)} = \text{one of} \begin{cases} d \\ pd \\ \log(p)d \\ p^2 d \end{cases}$$

Thus a total of 12 curve fits are performed using every combination of the startup and data functions and the best fit is selected. These functions were chosen as they provide reasonable fits for all cases thus far encountered. This approach gives simple functions with three parameters, e.g.

$$t = 14 + 123 * \log(N_{procs}) + 1.2 * N_{data} * \log(N_{procs})$$

There is an issue of the number of significant figures which should be quoted. An estimate of the standard error of the parameters may be made if the error of the measurements is known, and this can be used to give expected minimum and maximum times (t_min and t_max equations). This standard error may be used to guide the accuracy of quoted figures. An alternative, justified because it makes quick rough-and-ready calculations easier, is to round the coefficients to the nearest order of magnitude, so the above equation becomes:

$$t = 10 + 100 * \log(N_{procs}) + 1 * N_{data} * \log(N_{procs})$$

An attractive way to do this is to use the logarithm of the time (in μs) rounded to the nearest integer. This gives buckets at $1\mu s$, $10\mu s$, $100\mu s$, $1ms$, $10ms$ etc. which allows quick and simple performance estimates to be done, accurate to within an order of magnitude.

3.2.3 Categorisation of functions into "buckets"

Functions could be placed into one of two categories - "Quick" ones and "Slow" ones, with some threshold used to differentate between the two (say $100\mu s$). This has the advantage of being very simple to use but lacks the subtlety required.

3.3 Measuring MPI performance

Characterising the behaviour of the MPI functions is straightforward in principle; measure the time to complete N calls and take the average. The parameters of interest are the number of processors and the size of the messages.

To time an operation (e.g. `MPI_Bcast()`), a short function is written:-

```
void time_Bcast(int numelems, int iter, double &time)
{
  int *buffer = new int[numelems];
  MPI_Barrier( comm );
  double e1 = MPI_Wtime();

  // Operation to time
  MPI_Bcast( buffer, numelems, MPI_INT, 0, comm );

  time = MPI_Wtime() - e1;
```

```
    time = getmax( time );
    delete buffer;
  }
```

The `MPI_Wtime()` function is used to time the operation. The processes are synchronised beforehand using an `MPI_Barrier`. This is not perfect, as some processes may return from the barrier before others, so an alternative synchronisation technique has also been used which first determines the clock skew between different processes' `MPI_Wtime()` values, then busy waits until the timer reaches an agreed value. This provides synchronisation to a resolution of the short time required to read the timer, but just using `MPI_Barrier` is more convenient in practice.

The time is measured from this synchronisation point until the last process has returned. The `getmax()` function uses an `MPI_Reduce` across all processes to determine this maximum delay and the resolution of the timer is reported from the `MPI_Wtick()` function.

The parameters are the size of the message and the number of processes in the current communication group `comm`. These are varied across the range of values of interest on the machine, and each timing is repeated to produce a 3D set of measured times of the operation on the machine.

A separate `time_...` routine is needed for each MPI function. The routine:-

```
void time_routine(
  op_fn op,
  char *fname,
  int vary_nprocs,
  int vary_nelems
);
```

calls the basic operation timing function, varying the number of processors and data size. Timings are repeated `niters` times to obtain the minimum, maximum and median values at each data point. (The median is used in place of the mean as it is less susceptible to outlying points).

Operations fall into four classes -

- point to point routines (vary data size),
- collective routines (vary data size and number of processors),
- barrier-type routines (vary number of processors) and

- local operations (vary nothing).

For blocking point to point operations, the time for the send or receive to complete is taken, where both sender and receiver have been synchronised with a barrier beforehand. It is also necessary to measure the receive time of a message which has already arrived. This is obtained by delaying the call to `MPI_Recv` by more than the expected time to transmit the message. In figure 3.3 **Tsend** is the time the sender is delayed; **Trecv** is the time the receiver is delayed when the send and receive start at the same time. **Trecvmin** is the best possible receive time, where the message send was deliberately started in advance in order to hide the network latency. With these values, it is possible to calculate the expected delays of any send/receive combination.

For the non-blocking operations, two times are recorded; the time to post the send or receive and the time for the subsequent `MPI_Wait()` to complete. Another figure which is useful is the time for which useful work may be performed before calling `MPI_Wait` without imposing any extra delay. This is measured by using `MPI_Test` to poll the request. Figure 3.4 shows the times measured for non blocking point to point routines. **Tisend1** is the time to post an asynchronous send. **Tisend2** is the time for the asynchronous send to complete. **Tisendoverlap** is the amount of time available for hiding computation between posting the send and it completing. Similarly **Tirecv1** is the time to post a receive, **Tirecv2** is the time for it to complete and **Tirecvoverlap** is the amount of useful work which can be hidden.

These point to point measurements are taken using just two processes on a quiet machine. They do not take message contention into account. To incorporate message contention, variants of the point to point measurements are also taken where half of the processes send to the other half. This leads to more pessimistic estimates of point to point communication times. A random permutation is used to select which pairs of processes communicate. It would also be possible to take measurements of particular point to point patterns such as neighbour exchanges on meshes. The random pattern was chosen in preference to this as it acts as a likely average time for such communications.

Collective operations will take different times for the different processes involved (figure 3.2), but to make the problem tractable the time measured is the time between the first process starting and the last process finishing. Some operations have more parameters than just the number of processes in the group and the data size. For `MPI_Comm_split`, there is a choice of parameters to measure, since a group may be split in any fashion according to the key. The one measured

here is the time to split a group of size N into two. For `MPI_Reduce`, the time for the reduction function ought to be part of the equation but initially only the standard addition operation is used.

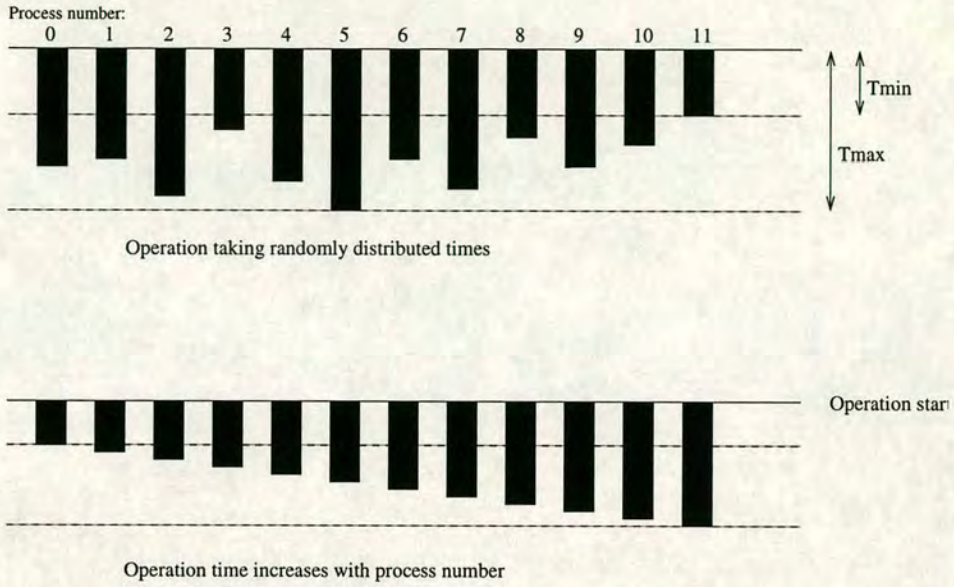


Figure 3.2: Possible discrepencies between collective timings made on different processes.

Detailed characterisation of the performance of *subgroups* was not carried out, as all of the test routines just made use of the global group of all processes `MPI_COMM_WORLD`. However, characterising the performance of these subgroups would be an interesting extension as it would quantify the benefits which could be expected from exploiting the (hopefully) faster performance of smaller, logically independent groups of processes.

One simple way of incorporating this into the measurements would be to repeat each collective measurement for a group of size P with an equivalent measurement using two groups each of size $P/2$ concurrently. The way in which the group is split is likely to have an effect on the performance, so it would be necessary to perform the obvious splits (randomly selected, first and second half, alternate). This would only give information for splitting a group into two. Repeat measurements would be needed for four, eight etc., giving a large amount of information in the datasheet.

Characterising the general case, where different subgroups perform entirely different operations, is difficult and in practice it would probably be simplest to use the models of the supergroup instead. Further work would be required to devise a usable general model for independent subgroups.

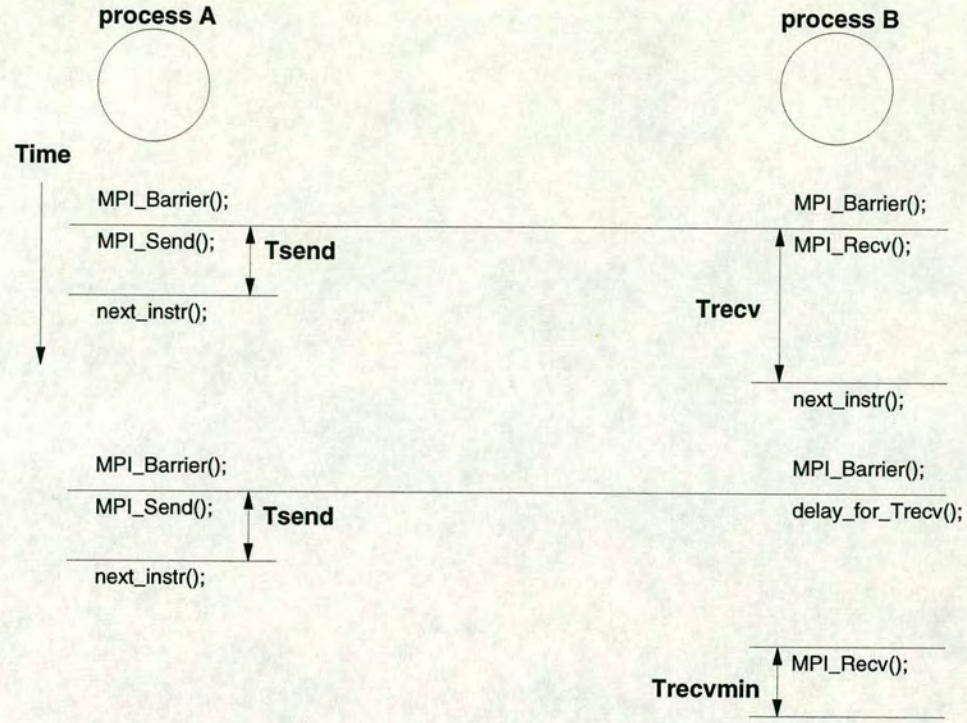


Figure 3.3: Times measured for blocking point-point operations.

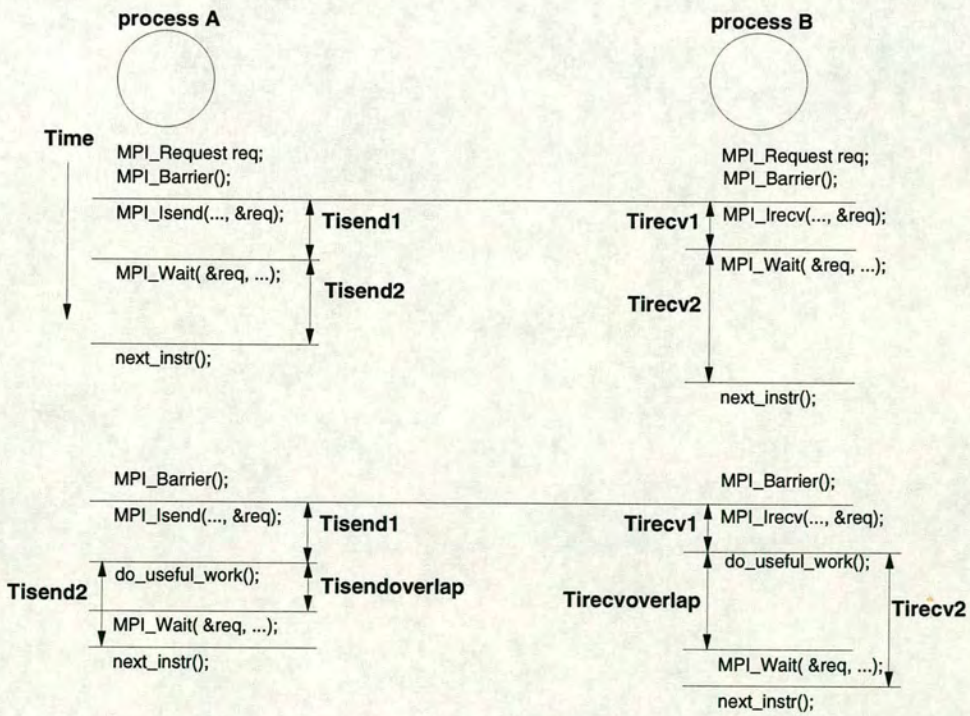


Figure 3.4: Times measured for non-blocking point-point operations.

3.4 Pitfalls

The timings have to be run when no other users are loading the system. These can seriously affect the results, by several orders of magnitude. This also affects the validity of predictions made using the performance models of the building blocks, since they will be unreliable on a loaded system.

There will also be variability on an unloaded system caused by interrupts, disk accesses not taking a uniform time and virtual memory paging. This may be incorporated into the measurements by repeating each timing and noting the spread.

On some machines, the time for message passing operations may vary by orders of magnitude depending on where the message happens to be in the memory hierarchy. This is a very hard variation to estimate in practice, since it depends on where the compiler puts data, the exact sizes of the various levels of caches, the buses between caches, the main memory system organisation and access time, and whether the data in the message has been recently accessed.

Rather than attempt to solve the Sisyphan problem, a decision was made to use freshly allocated memory for all timings of the routines. Since newly allocated memory will not have been read from by the timing program, it will reside in main memory rather than cache. Thus all MPI time measurements given are for the case where the data has not been cached, and may be more pessimistic than is observed in actual programs.

It would be possible to repeat all measurements for the case where the messages are cached, increasing the complexity of the data sheets. The question then arises: how will the user know which model to use? Without performing a full system simulation, it is very difficult to tell.

3.5 Data sheet generation

The generated data sheets give a page for each MPI function measured. Examples are given in section 3.6 below. Each page has the sections:

- Operation
- Full timing graph
- Fitted curve

In addition, there are summary tables at the start. The source code for the routines is given in [17]; the sections below describe how data sheets are generated.

3.5.1 Curve fitting

To obtain an equation summarising the performance of each function from the measured data, a linear least squares technique is used. The result is a curve for point to point operations and a surface for collective routines. As explained above, the form of the equation to fit is:

$$t = \text{c\_coeff} + \text{s\_coeff} * [\text{start\_mode}] + \text{d\_coeff} * [\text{data\_mode}]$$

where start_mode is one of (nprocs, log(nprocs), nprocs^2) and data_mode is one of (ndata, ndata $*$ nprocs, ndata $* \mathit{log}$(nprocs), ndata $*$ nprocs^2).

The least squares curve fit is run using all combinations of start_mode and data_mode to see which fits best (i.e. has the smallest chi-squared value).

The "goodness of fit" parameter Q is also recorded to give an indication of how accurately the equation fits the measured data. The standard error of each parameter yields equations giving the maximum and minimum expected times. This should only be used as a rough guide, as there is no guarantee (or even likelihood) that the measured data conforms to a normal distribution. However, it is useful to have at least some indication of expected confidence intervals.

3.5.2 Implementation of the surface fit routine

The input to the curve fitting routine is a set of measurements giving the group size, the data size, the average time and an estimate of the error of each measurement. The results are the coefficients of the equation, with estimates of the errors for each coefficient, e.g.

$$T_{bcast}(\mu s) = (100 \pm 10) + (6 \pm 0.8) \times nprocs + (0.04 \pm 0.002) \times nprocs \times ndata$$

To perform the curve fit, it is convenient to write the equations in matrix form and then apply a matrix solver to obtain the coefficients:

$$y(\mathbf{x}) = \sum_{k=1}^{M} a_k X_k(\mathbf{x})$$

where $\mathbf{x}$ is the vector of input parameters $(nprocs, ndata)$ and $X_1(\mathbf{x}), ..., X_M(\mathbf{x})$ are the functions of $\mathbf{x}$.

The merit function (χ^2) is

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{k=1}^{M} a_k X_k(\mathbf{x}_i)}{\sigma_i} \right]^2$$

From the design matrix the *normal equations* may be derived. These are solved using Gauss-Jordan elimination to give the vector of coefficients, the chi-squared value and the vector of coefficient variances.

This process is repeated for all possible combinations of the X_k functions and the combination which yields the lowest chi-squared value is chosen.

3.5.3 Units and Significant Figures

The simplest unit of time is absolute time for a specific architecture. Unless there are real time constraints, however, it is more useful to know a time relative to the time to perform arithmetic (or memory access) operations. The times could be given in processor cycles, but this wouldn't mean *that* much because of the memory hierarchy. Hence times are given in milliseconds or microseconds (depending on a compile time flag).

The appropriate number of significant figures to quote in the datasheet is debatable. One extreme is to round all coefficients to the nearest order of magnitude. This makes rough and ready estimates of a function's speed very straightforward. The other extreme is to use as many figures as are justifiable given the standard errors of the coefficients. The solution chosen was to let the user choose the number of significant figures at compile time.

3.5.4 Output formats

The datasheet generation is performed by two routines;

- `rawtiming` is an MPI program which gathers the data, and
- `dsheet` performs the curve fitting and generates the latex document.

The split into a data gathering program and a data analysis routine makes performing several analyses on the same data more convenient than if the two routines were combined. Figure 3.5 shows the connections between the routines.

3.5.4.1 Output from `rawtiming`

For each operation measured, two data files are produced; one for large message sizes and one for small message sizes. The reason for separating small and large message sizes is that there is often a nonlinearity in the performance curve. Performing separate fits for small and large messages makes for more accurate equations. The threshold between small and large messages may be varied; by default it is set at 32 integers, or what may reasonably be regarded as parameters

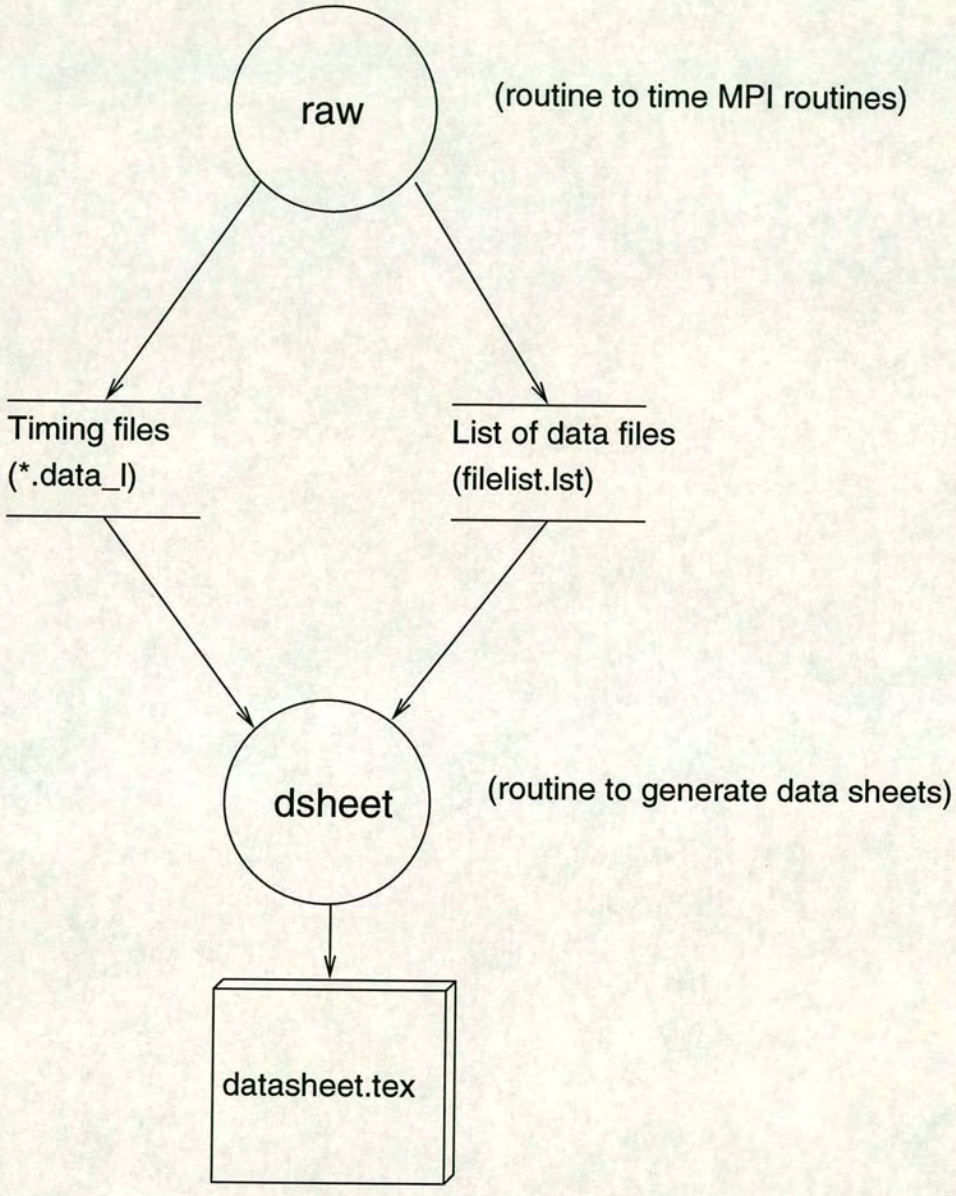


Figure 3.5: Datasheet generation routines.

rather than a block of data. For example, the two data files for the `allgather` operation are called:

```
allgather.data_l
allgather.data_s
...
```

The format of the data files is

```
2 32 0.000138214 0.0000045
2 64 0.000146659 0.0000043
2 128 0.00016432 0.0000041
2 256 0.000201307 0.0000035
2 512 0.000270488 0.0000065

4 32 .... etc
```

with the columns storing the number of processes, the message size, the median time taken and an estimate of the measurement error.

The file `filelist.lst` is also generated; it stores the parameters, date on which the measurements were taken, and a list of the routines which were measured. It is used by the `dsheet` program to generate the datasheet.

3.5.4.2 Output from `dsheet`

The `dsheet` program turns the raw data into a latex file `datasheet.tex` along with a set of GNUplot scripts for displaying the measured data and the approximated curve fits. It expects the file `filelist.lst` and all the raw data files to be in the current working directory.

3.5.5 A simple calculating utility

As an alternative to using the datasheets by hand, a simple tool was developed to evaluate the equations for a given operation, data and group size.

The routine reads in the results of `dsheet` and presents a simple command line interface. A sample session is:

```
[balnagowan]fwh: timecalc summary.lst

Simple utility for estimating the time of MPI
functions based on the summary.lst file produced by
```

```
rawtiming.
Enter <name> <groupsize> <datasize>

>
> bcast 16 1000
# t = (0.000106549 +/- 1.23071e-05) +
#     (6.35065e-06 +/- 7.83058e-07)*nprocs +
#     (4.39693e-08 +/- 1.75882e-09)*nprocs*ndata
Times (s): min=0.00086141 avg=0.000911668 max=0.000961926
>
> alltoall 16 1000
# t = (1.41845e-05 +/- 4.80534e-07) +
#     (4.61065e-05 +/- 2.9256e-06)*nprocs +
#     (2.44134e-07 +/- 8.85081e-09)*nprocs*ndata
Times (s): min=0.00444749 avg=0.00465803 max=0.00486857
```

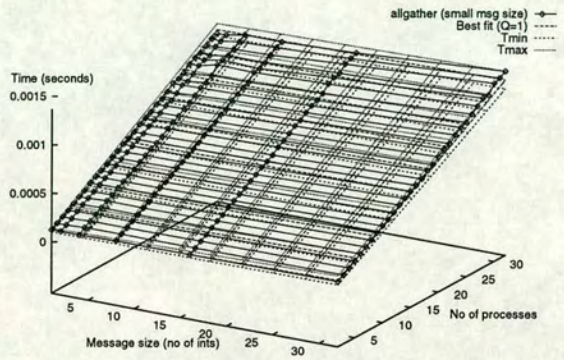
3.6 Example data sheets

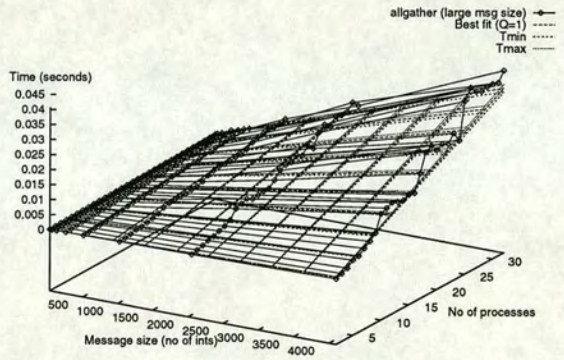
Table 3.1 shows an excerpt from the summary part of a data sheet (for the Cray T3D); a detailed page is shown in figure 3.6. A complete data sheet document is included in appendix B.

3.7 Conclusion

This chapter has presented a routine for characterising the performance of MPI functions. The approach is a compromise between usability and accuracy. The aim has been to allow design for specific architectures based on measured data rather than guesswork. Computer based design tools using this data, as an alternative to pencil and paper, are the subject of the next three chapters.

allgather





$$T_{allgather}(\mu s) = \begin{cases} (50 \pm 20) + (40 \pm 1) \times nprocs + (1 \pm 0.9) \times ndata & \text{if } ndata <= 32 \\ (4 \pm 20) + (40 \pm 3) \times nprocs + (0.3 \pm 0.009) \times nprocs \times ndata & \text{if } ndata > 32 \end{cases}$$

19

Figure 3.6: A page from an automatically generated MPI data sheet.

MPI Function	Expected time (μs)			Goodness of fit (Q)
send	$70 +$	$3 \times d$		0.61
ssend	$100 +$	$2 \times d$		0.84
rsend	$70 +$	$3 \times d$		0.6
isend1	$70 +$	$3 \times d$		0.47
isend2	$20 +$	$0 \times d$		0.93
isendoverlap	$3 +$	$0 \times d$		0.97
recv	$70 +$	$5 \times d$		0.46
recvmin	$50 +$	$3 \times d$		0.43
irecv1	$30 +$	$0.4 \times d$		0.96
irecv2	$60 +$	$3 \times d$		0.57
irecvoverlap	$0.5 +$	$1 \times d$		0.043
sendrecv	$100 +$	$5 \times d$		0.57
pingpong	$200 +$	$8 \times d$		0.52
alltoall	$10 +$	$50 \times p +$	$0.2 \times p \times d$	1
gather	$80 +$	$8 \times p +$	$0.07 \times p \times d$	1
allgather	$4 +$	$40 \times p +$	$0.3 \times p \times d$	1
reduce	$200 +$	$10 \times p +$	$0.6 \times log(p) \times d$	1
allreduce	$300 +$	$20 \times p +$	$0.9 \times log(p) \times d$	1
bcast	$100 +$	$6 \times p +$	$0.04 \times p \times d$	1

Table 3.1: Summary table for Small messages (32 integers or less). p is the number of processors in the group and d is the message size (number of integers)

Chapter 4

Simple Performance Estimates of the Cowichan Problems

The data sheets described in the previous chapter may be used as they are for manual estimates of performance and scalability. However this becomes impractical for all but the smallest programs, so this chapter looks at a way of feeding the datasheets into an equation and graph plotting program. This study was performed to investigate when such a minimalist approach to modelling may be applied.

Section 4.1 describes the basics of the approach; section 4.2 shows how data sheet results are integrated; section 4.3 describes the models of the Cowichan problems and section 4.4 discusses the results.

4.1 The models

Simple performance models of the MPI Cowichan suite were constructed using the GNUplot package. GNUplot turned out to be a powerful tool, providing functional composition to express hierarchy, as well as providing graphs.

The aim of this work was extremely rapid construction of models with gross assumptions to see how valuable such an approach can be, contrasted to more accurate (and time consuming) modelling.

In the simplest version, computation and communications costs were denoted by parameters $T_{compute}$ and T_{comms} (in section 4.2 the single communications parameter is replaced by the set of equations from a data sheet). Collective operations were modelled using simple combinations of the parameters, e.g.

$$log(N_{procs}) * T_{comms} * N_{data}$$

The computation and communication performance parameters for all models are specified in a GNUplot script file:

```
tcomms            = 10.0
tcompute          = 1.0
tbcast(p,d)       = tcomms*d*log(p)
tallreduce(p,d)   = tcomms*d*log(p)
talltoall(p,d)    = tcomms*d*log(p)
talltoallv(p,d)   = tcomms*d*log(p)
talltoallvi(p,d)  = tcomms*d*p
tallgather(p,d)   = tcomms*d*log(p)
tallgatherv(p,d)  = tcomms*d*log(p)
```

This gives the expected time for the communications functions `MPI_Bcast`, `MPI_Alltoall` etc. in terms of the data size **d** and the number of processors **p**. The parameters `tcomms` and `tcompute` give the expected times in microseconds to send an integer or perform a computation step.

Individual models include these top level parameters.

4.1.1 An example: the mandelbrot set

The model for the mandelbrot set example appears as:

```
# model of mandel performance
tmandelcalc       = tcompute * 8 * maxmandeliter
tmandel(p)        = ncols * (nrows/p) * tmandelcalc
```

`tmandelcalc` is the time required to compute a single pixel of the set, given here as eight compute steps per iteration multiplied by the maximum number of iterations. `tmandel(p)` is the estimated time to compute the set on **p** processors where each processor has an equal slice of the matrix to work on (i.e. a slice of size $ncols * (nrows/p)$).

Because these equations are included in a script file for GNUplot, a graph (figure 4.1) may be produced with the line:

```
plot tmandel(x)
```

This just models the computation to be done, but the communication must also be accounted for. This leads to a model such as:

```
# model of mandel performance (2)
tbroadcast(p)     = tbcast(p,8)
tgatherresults(p) = tgather(p,ncols*nrows/p)
tmandelcalc       = tcompute * 8 * maxmandeliter
```

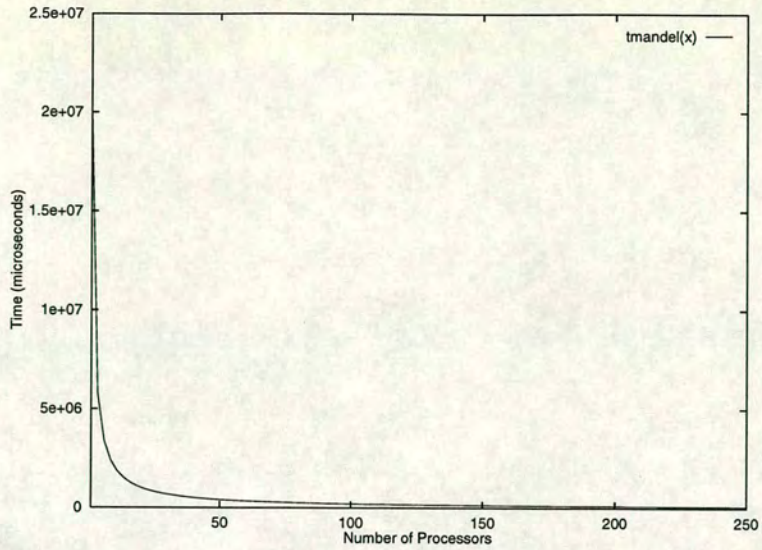


Figure 4.1: Simple plot of expected mandelbrot performance.

```
tmandelcomp(p)      = ncols * (nrows/p) * tmandelcalc
tmandel(p)          = tbroadcast(p) + tmandelcomp(p) + tgatherresults(p)
```

The components are shown in figure 4.2, with `tcomm` set to 1000.

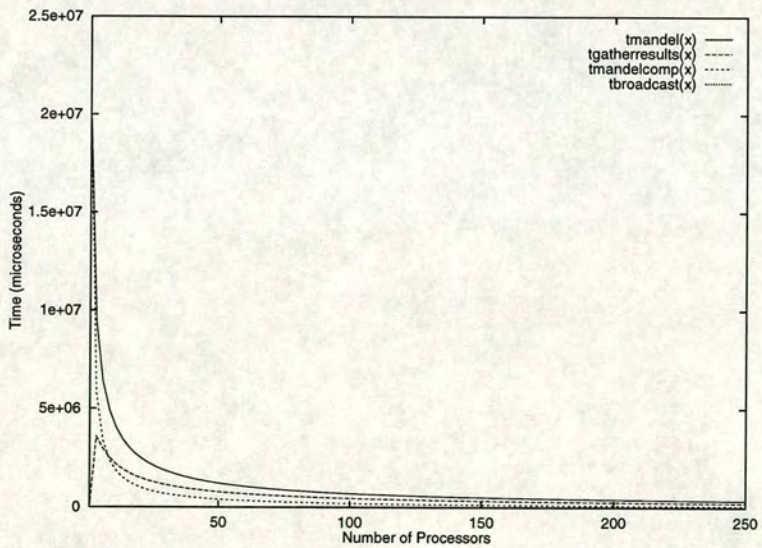


Figure 4.2: Time plot of expected mandelbrot performance with communications added.

Speedup curves may also be plotted (figure 4.3):

```
plot tmandel(1) / tmandel(x)
```

Figure 4.4 shows the speedup curve with the compute times stepped from 1.0 to 100.0 and the communications time left at 1000.0.

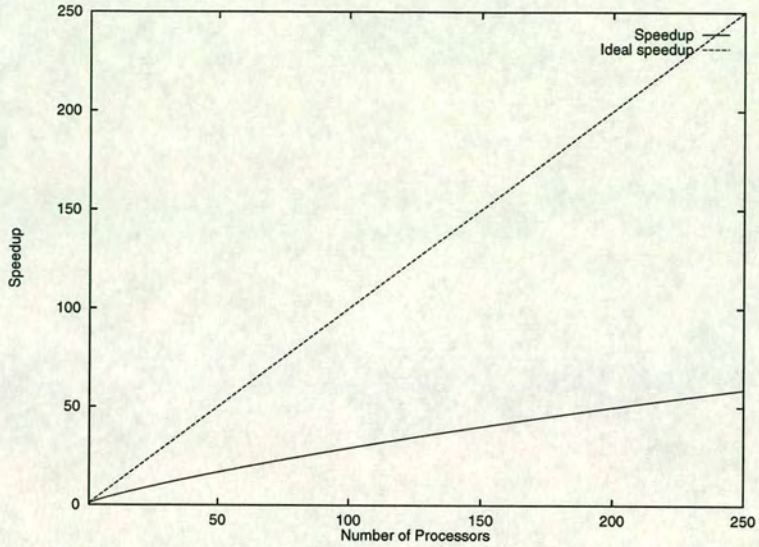


Figure 4.3: Speedup plot of expected mandelbrot performance with communications.

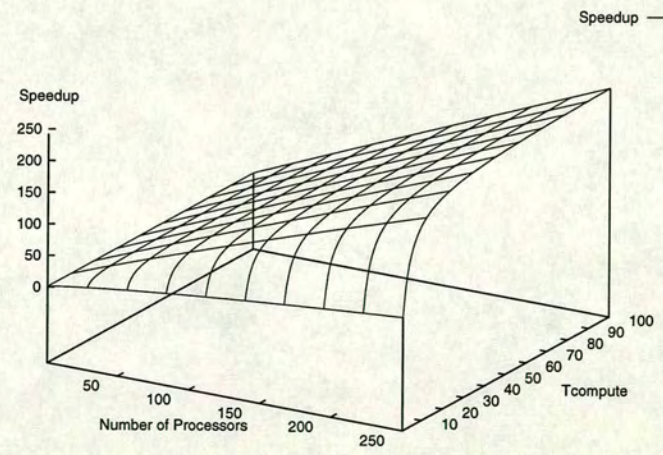


Figure 4.4: Speedup plot: `tcompute` varied from 1 to 100.

All of these graphs may be produced extremely rapidly from a basic model of a parallel program. Given particular values of `tcompute` and `tcomms` it is straightforward to determine whether a particular algorithm will scale well on a parallel machine. The difficulty is knowing what values to use for `tcompute` and `tcomms` for an architecture. The simplistic estimates of collective performance used above are not very believable, so a method using models derived from data sheets is described in the next section.

4.2 Using data sheet models

The MPI datasheet generator described in chapter 3 produces a set of equations characterising the MPI communications performance.

The measured model for the Cray T3D is shown below (times are in microseconds; `p` is the number of processors in the group and `d` is the message size).

```
# Cray T3D model
talltoall(p,d)   = 40*p + 0.3*p*d
tallsend(p,d)    = 40 + 0.1*p + 0.1*d
tgather(p,d)     = 200*log(p) + 0.0009*p*p*d
tallgather(p,d)  = 40*p + 0.3*p*d
treduce(p,d)     = 300 + 2*p + 0.6*log(p)*d
tallreduce(p,d)  = 300 + 6*p + 1*log(p)*d
tbcast(p,d)      = 100 + 2*p + 0.2*log(p) *d
tbarrier(p)      = 40
```

For comparison, the model for a network of 8 Sun SPARCstation 5 workstations connected using ethernet is:

```
# Network of Workstations model
talltoall(p,d)   = 2000 + 4000*p*p + 2*p*p*d
tallsend(d)      = 2000 + 1*d
tgather(p,d)     = 30000*log(p) + 4*log(p)*d
tallgather(p,d)  = 8000*p + 4*p*p*d
treduce(p,d)     = 20000*log(p) + 6*log(p)*d
tallreduce(p,d)  = 10000 + 500*p*p + 2*d*p*p
tbcast(p,d)      = 2000 + 700*p*p + 1*p*d
tbarrier(p)      = 9000*p
```

This may be incorporated into the performance models for programs, effectively instantiating measured values for `tcomm`. Figure 4.5 shows the expected

speedups using the Cray model for communications and varying `tcompute` from 0.01 to 1.0. Note that a *slowdown* is expected at low values of `tcompute`; figure 4.6 shows that the minimum time occurs at 70 processors. Figure 4.7 shows the expected speedup using the network of workstations performance model.

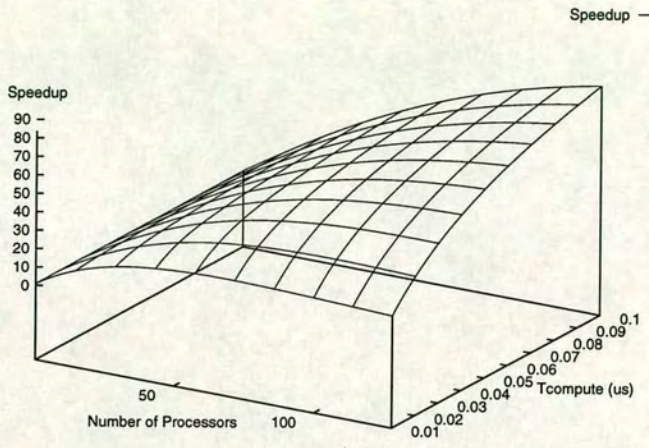


Figure 4.5: Speedup plot generated using datasheet model for Cray T3D performance.

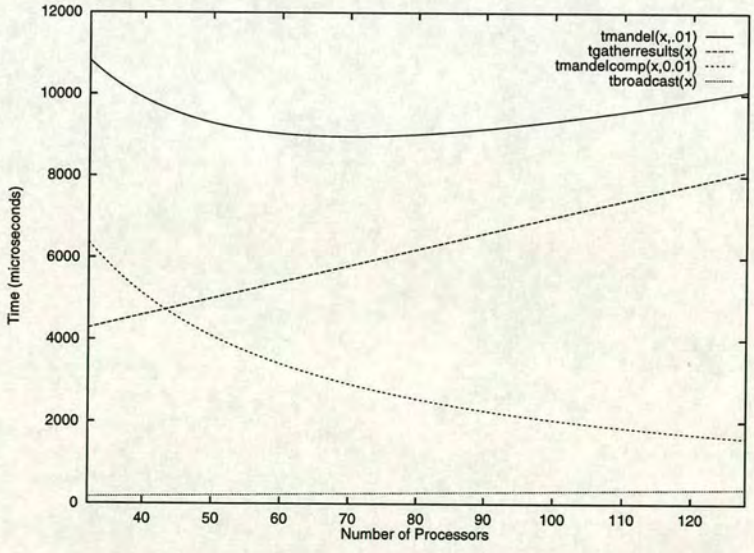


Figure 4.6: Closeup of figure 4.5 at `tcompute=0.01us`.

4.3 Modelling the Cowichan problems

This section describes models of all the Cowichan problems detailing what design information may be gleaned from using this design technique. The models are

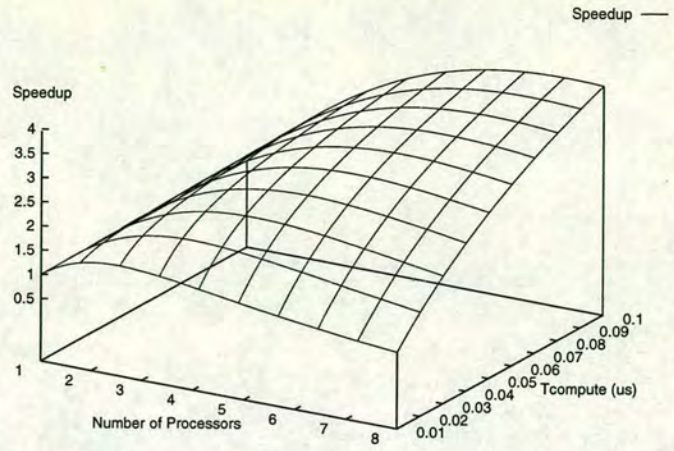


Figure 4.7: Speedup plot generated using datasheet model for Network of Workstations performance.

presented in GNUplot equation format, with an equation for the time of each phase of the program in terms of the number of processors **p** and the data size **d**.

4.3.1 Mandelbrot set generation (`mandel`)

Each of the points in the mandelbrot set may be computed independently, which makes this an "embarrassingly parallel" problem. The only difficulty for performance prediction is that the computation at any point is unpredictable, with any number of iterations in the central computation loop between 1 and the maximum number of iterations `MAX_MANDEL_ITERS`. The black regions inside the set require large amounts of computation, those far from the set require very little. The model given below includes the parameters **p** (the number of processors), **d** (number of rows in the matrix) and **c** (the compute step time, set at $1us$).

```
# model of mandel performance
tmandelcalc(c)      = c * 8 * maxmandeliter
tmandelcomp(p,d,c) = d * (d/p) * tmandelcalc(c)
tmandel(p,d,c)      = tmandelcomp(p,d,c) + tbarrier(p)
```

Figure 4.8 shows the measured and predicted speedup on a network of workstations. The shapes of the curves match very closely; the predicted slowdown with small data sizes does actually occur, and the best speedup occurs at eight processors. Using sixteen processors yields no additional speedup as expected. Figure 4.9 shows the measured and predicted times which also match up well.

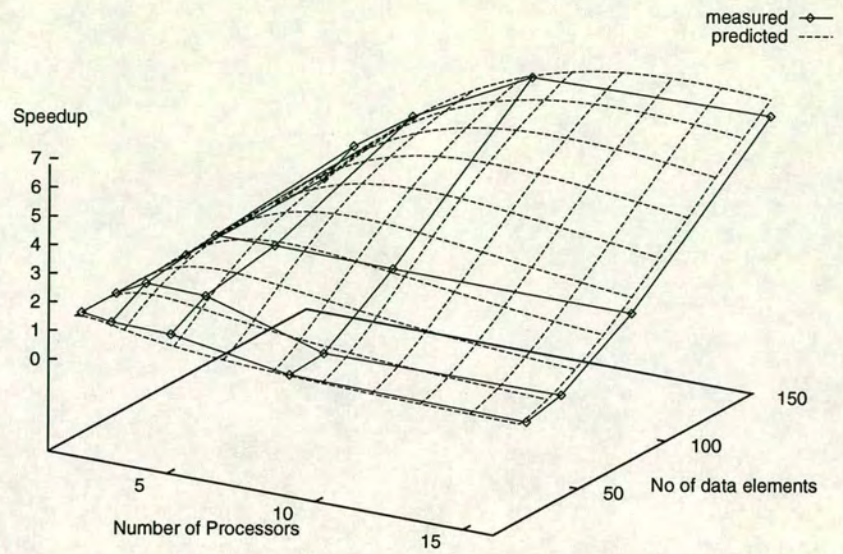


Figure 4.8: Mandel measured and predicted speedup on a network of workstations.

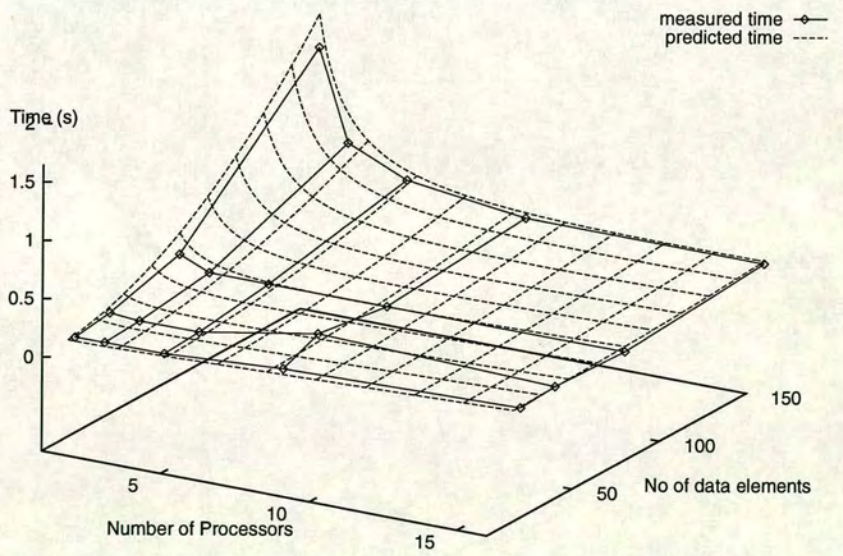


Figure 4.9: Mandel measured and predicted times on a network of workstations.

On the Cray T3D, with the compute time step left at $1us$ the measured and predicted times are shown in figure 4.10, with the speedup in figure 4.11. The speedup prediction is overly optimistic at low data sizes and overly pessimistic for high data sizes. The reason for this was determined by looking at the detailed timing diagrams. The matrix is distributed by entire *rows*. A 20×20 matrix size only makes use of 20 out of the available 32 processors, leaving the other 12 idle. This quantisation effect was not included into the simple model, resulting in the overly optimistic prediction. The second discrepancy was caused by missing an initial startup computation cost from the model - the time to allocate the memory for the matrix. This data dependent startup cost actually *improves* speedup with larger data sizes, since it has more impact on the single processor timing than on the multiple processor timing. This discrepancy was not apparent for the network of workstations comparison because communications time was dominant in that case.

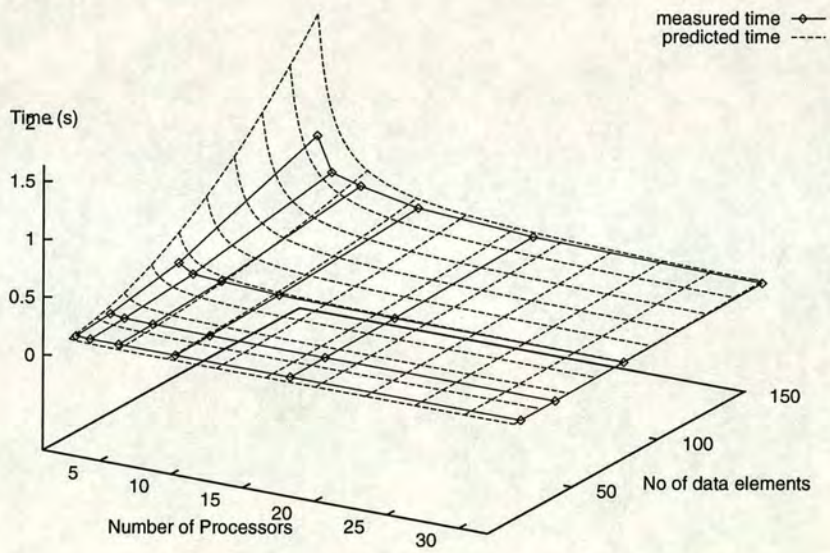


Figure 4.10: Mandel measured and predicted times on the Cray T3D.

4.3.2 Random matrix generation (`randmat`)

The matrix is divided equally among the processors and each computes the random numbers within its section. The overhead with respect to the sequential algorithm is that the initial seed must be computed for each process before it can start generating. This initial seed calculation requires a time proportional

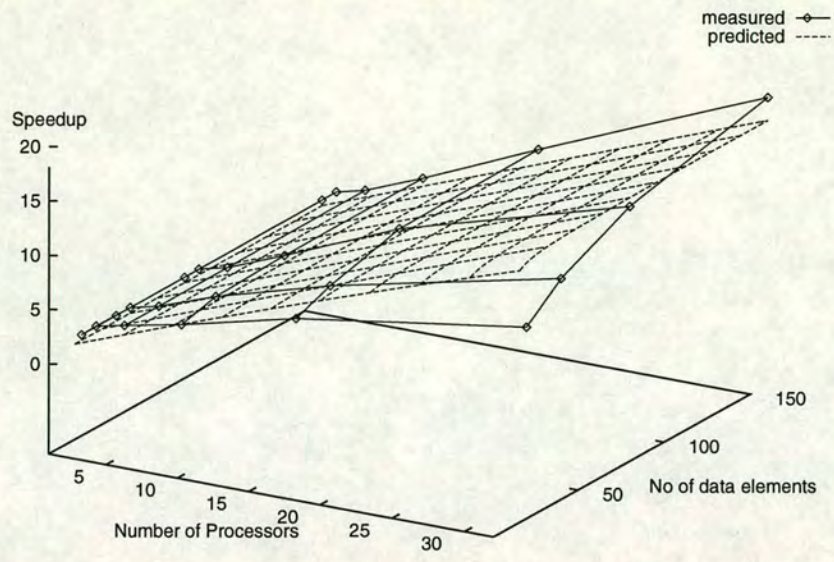


Figure 4.11: Mandel measured and predicted speedup on the Cray T3D.

to the logarithm of the number of matrix elements, so takes slightly longer for processors at the bottom of the matrix than for those at the top.

```
# model of randmat performance
tjrandom(i,p,d,c) = log(i*d*d/p) * c * 8
trandmat(p,d,c)   = c * d * d/p * 10
ttotal(p,d,c)     = tjrandom(p,p,d,c) + trandmat(p,d,c) + tbarrier(p)
```

Figure 4.12 shows the measured and predicted speedup on a network of workstations. All the important aspects of the performance are predicted correctly; the slowdown above four processors with the maximum data size, and the slowdown with more than one processor at the minimum data size. Figure 4.13 shows the measured and predicted times. For small data sizes, the time is dominated by the barrier time. The computation time has been consistently underestimated by a factor of two.

On the Cray, figure 4.14 shows the times (on a logarithmic axis). The prediction is an underestimate for small numbers of data elements but converges for larger matrix sizes, indicating that a constant overhead of the order of $500us$ has been left out of the model.

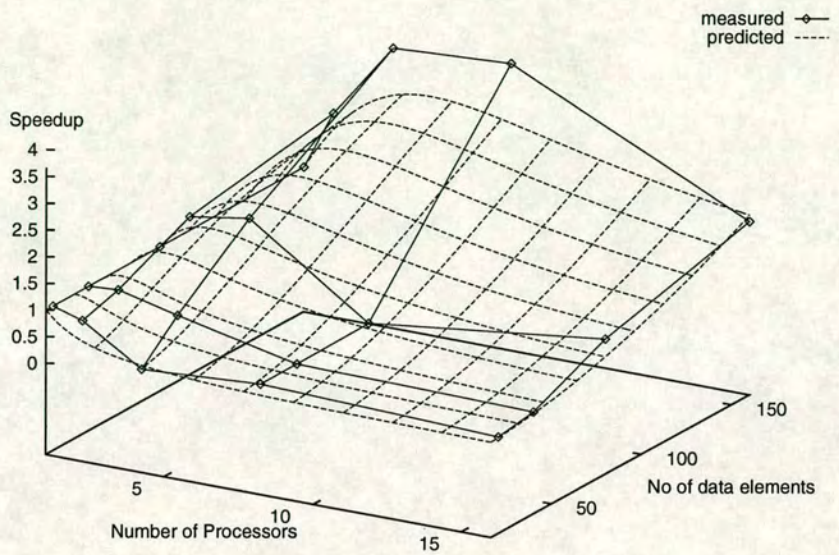


Figure 4.12: Randmat measured and predicted speedup on a network of workstations.

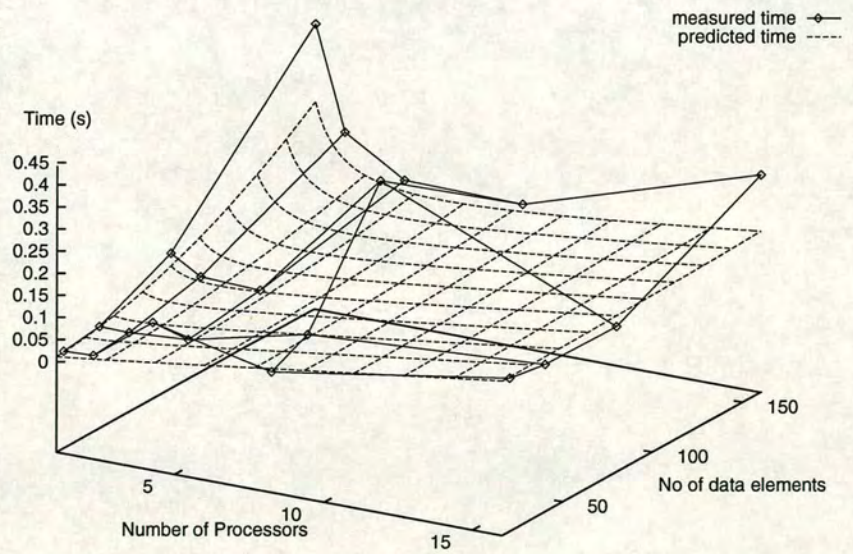


Figure 4.13: Randmat measured and predicted times on a network of workstations.

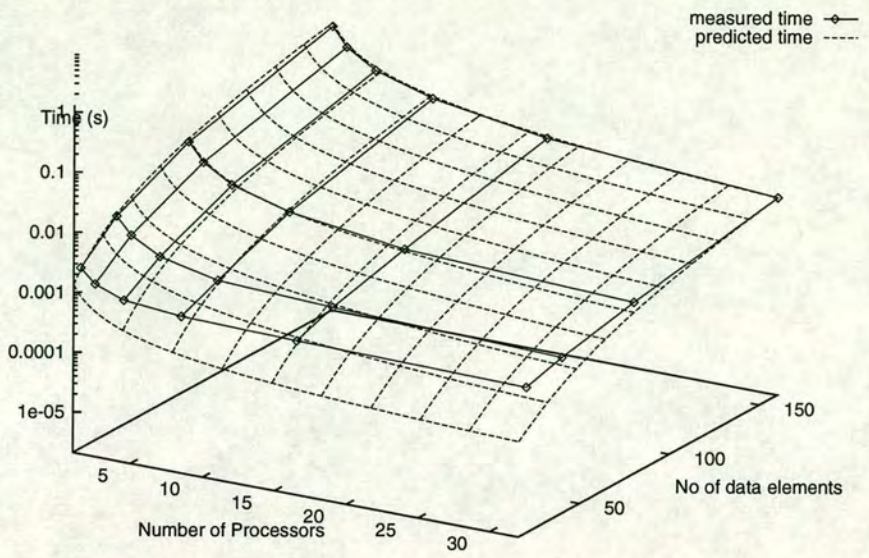


Figure 4.14: Randmat measured and predicted times on the T3D.

4.3.3 Perfect shuffle (`half`)

This module is heavy on communications as half of the matrix must be sent at each shuffle step. The model takes into account the time to build the complex MPI message data type required to complete the shuffle with one communications call.

```
# model of half performance
tcomputedtypes(p,d,c) = d * (d/p) * c * 2
tlocalshuffle(p,d,c)  = d * (d/p) * c * 3
ttotal(p,d,c)         = tcomputedtypes(p,d,c) + \
                        talltoallvi(p, (d/p)/p) + \
                        tlocalshuffle(p,d,c)
```

Figure 4.15 shows the measured and predicted speedup on a network of workstations. The prediction has yielded the useful information that a maximum speedup of two can be expected across eight processors. Figure 4.16 shows the measured and predicted times. These agree for all but the one point at the maximum number of processors and data.

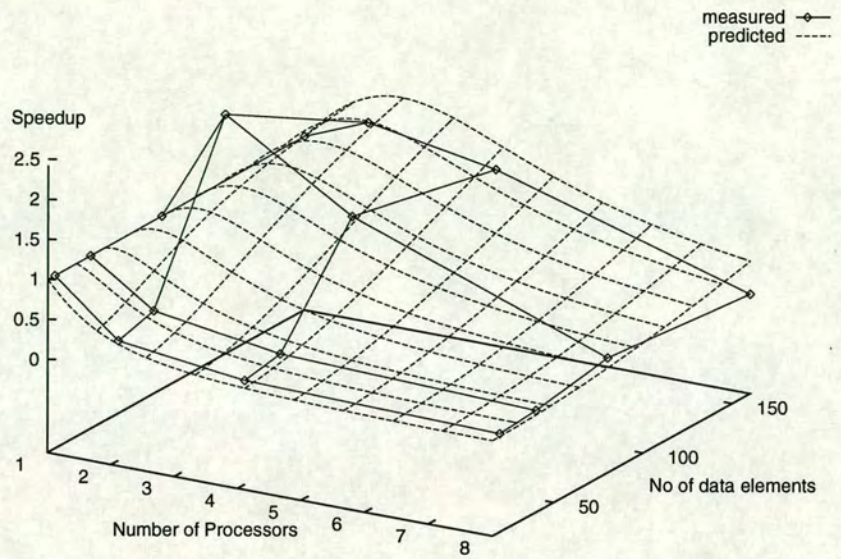


Figure 4.15: Shuffle (Half) measured and predicted speedup on a network of workstations.

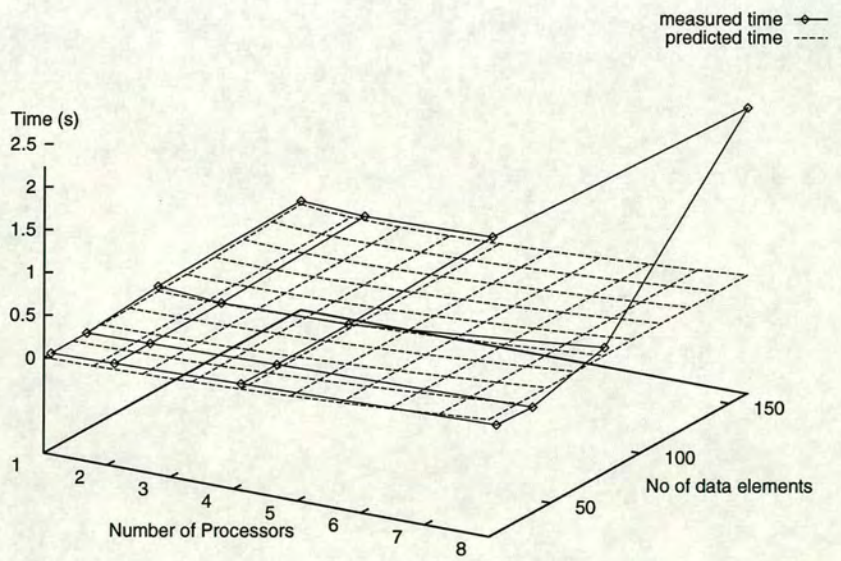


Figure 4.16: Shuffle (Half) measured and predicted times on a network of workstations.

4.3.4 The game of life (`life`)

This has a boundary swap followed by the local computation. The boundary swap consists of two sends followed by two receives.

```
# model of life performance
tboundaryswap(d)     = 2*tsend(d) + 2*trecv(d)
tliferow(d,c)        = c * d
tsublife(p,d,c)      = (d/p) * tliferow(d,c) + \
                       d * (d/p) * c
titer(p,d,c)         = tboundaryswap(d) + tsublife(p,d,c)
ttotal(p,d,c)        = nlifeiters * titer(p,d,c) + tbarrier(p)
```

Figure 4.17 shows the measured and predicted times on a network of workstations. As predicted, the times are dominated by communications, so no speedup is obtained. Figure 4.18 shows the measured and predicted speedups.

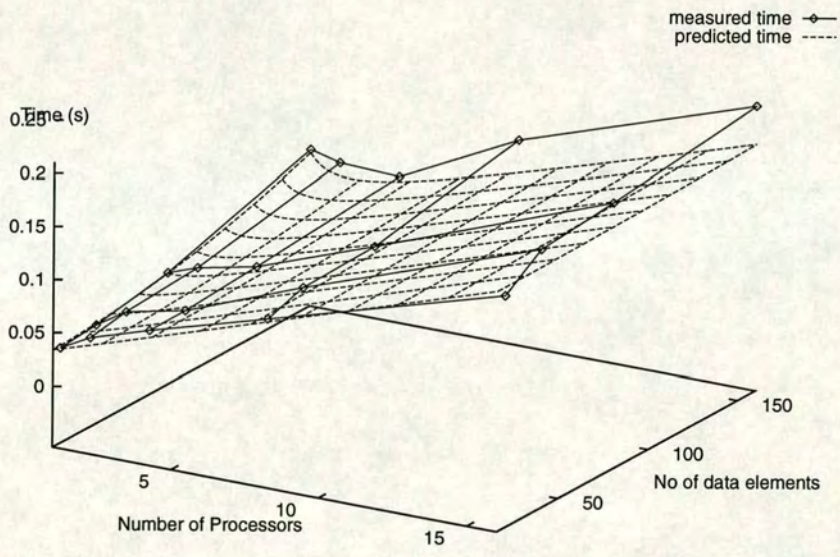


Figure 4.17: Life measured and predicted times on a network of workstations.

On the Cray, the measured and predicted times are shown in figure 4.19. The predictions are good for small numbers of processors, but out by a factor of three for 32 processors, because of quantization effects.

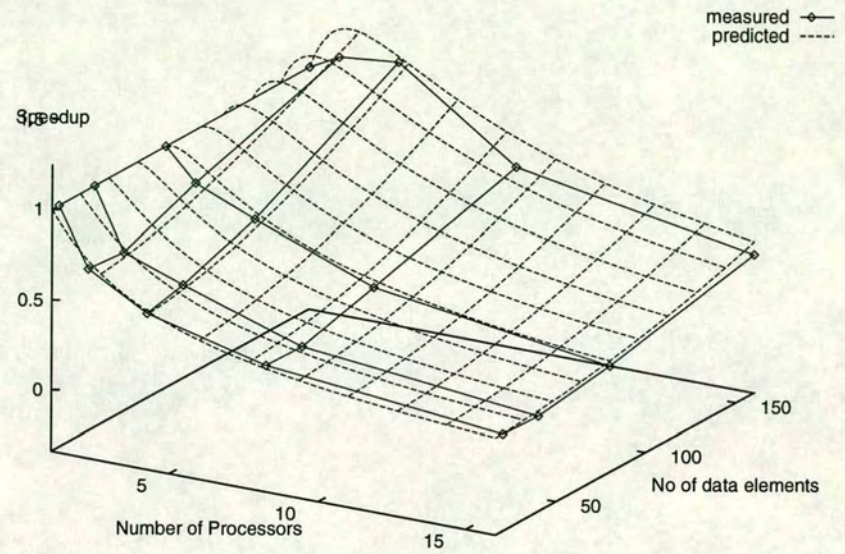


Figure 4.18: Life measured and predicted speedup on a network of workstations.

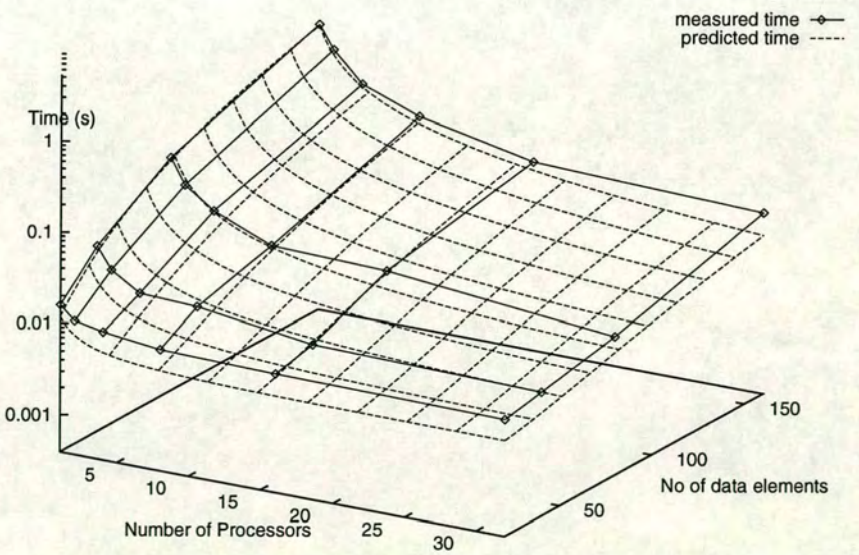


Figure 4.19: Life measured and predicted times on the T3D.

4.3.5 Image thresholding (thresh)

This model includes the local and global histogram computations time, the time to compute the threshold value and the time to generate the mask image.

```
# model of thresh performance
tglobalminmax(p,d,c) = d * (d/p) * c + tallreduce(p,1)
tlocalhist(p,d,c)    = c * d * d / p
tglobalhist(p)       = tallreduce( p, maxval )
tthresh(p,c)         = c * fraction * maxval
tmask(p,d,c)         = c * d * d/p

ttotal(p,d,c)        = tglobalminmax(p,d,c) + tlocalhist(p,d,c) +\
                       tglobalhist(p) + tthresh(p,c) + tmask(p,d,c) +\
                       tbarrier(p)
```

Figure 4.20 shows the predicted and measured times for a network of workstations, and figure 4.21 shows the predicted and measured speedups. This application shows an expected and measured slowdown on networks of workstations.

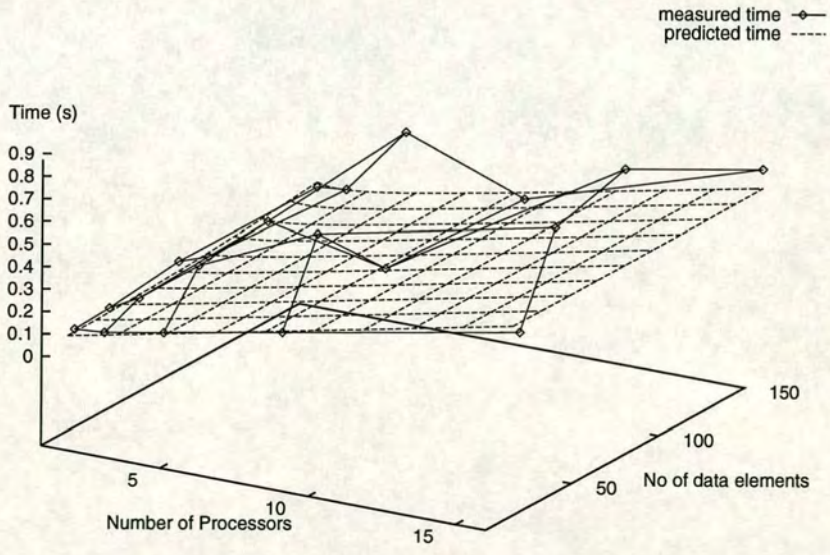


Figure 4.20: Thresh measured and predicted times on a network of workstations.

Figure 4.22 shows the measured and predicted times on the T3D. The times are underestimated by a factor of two; figure 4.23 shows that the speedup has been reasonably estimated.

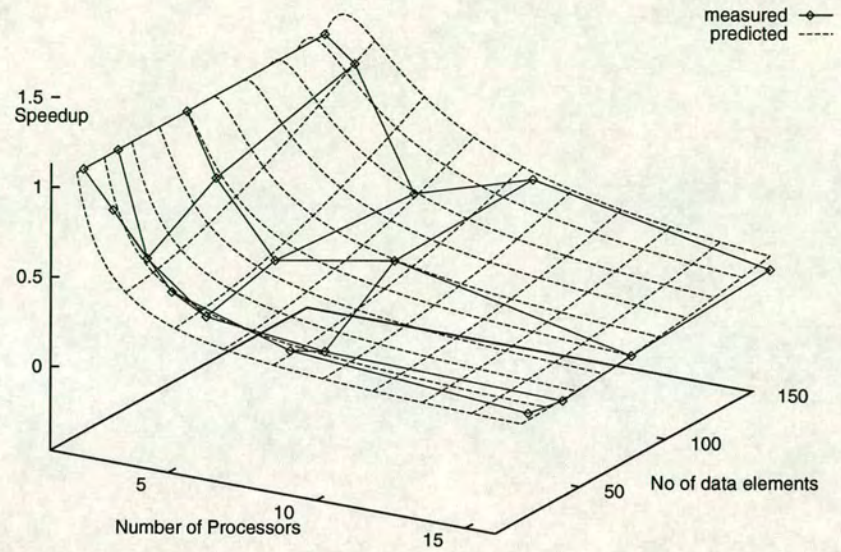


Figure 4.21: Thresh measured and predicted speedups on a network of workstations.

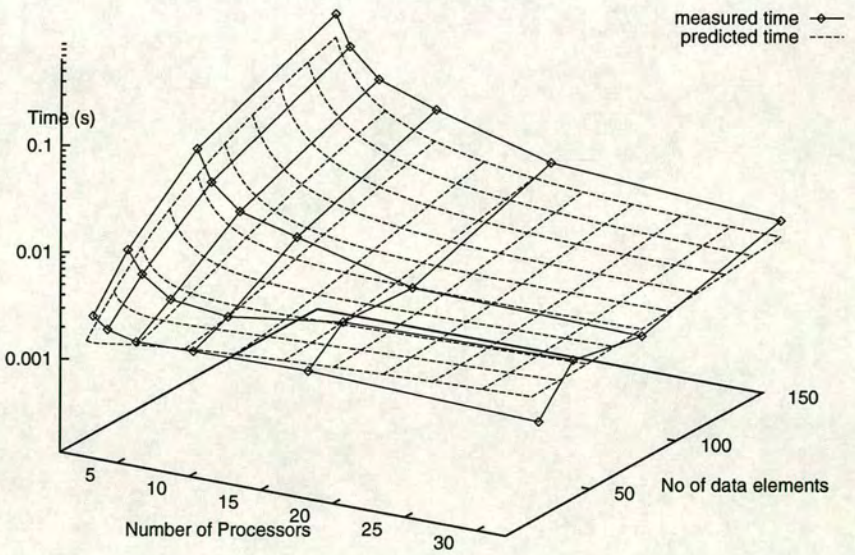


Figure 4.22: Thresh measured and predicted times on the T3D.

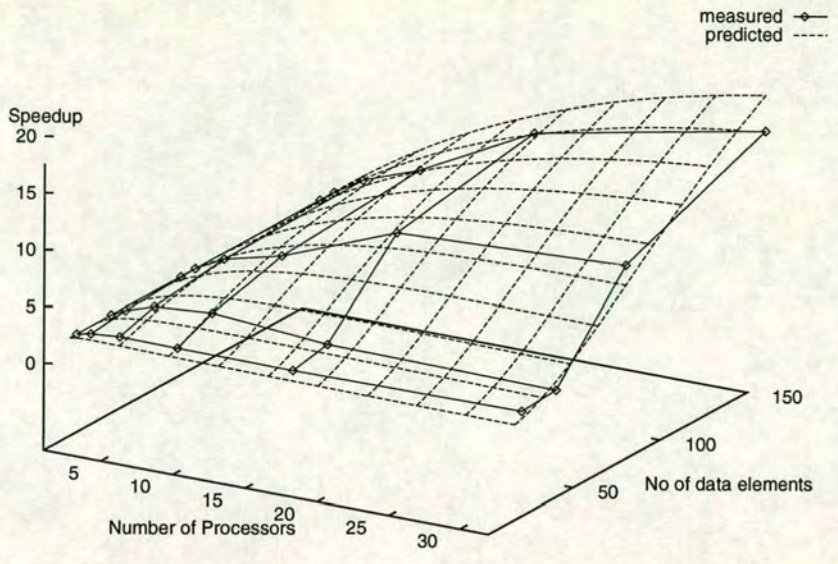


Figure 4.23: Thresh measured and predicted speedups on the T3D.

4.3.6 Outer product (outer)

This is interesting because of the possibility of cutting down the amount of computation by communicating results from the lower diagonal of the matrix to the upper diagonal.

```
tdist(c)              = 50*c
tmklocal(p,d)         = tallgatherv(p,d)
tcreate(p,d,c)        = d * tdist(c)
tfillmatrix(p,d,c)    = d * (d / p) * tdist(c)
tdiagfill(p,d,c)      = d * tdist(c)
ttotal(p,d,c) = tmklocal(p,npoints/p) + tcreate(p,d,c) +\
                tfillmatrix(p,d,c) + tallreduce(p,1) +\
                tdiagfill(p,d,c) + tbarrier(p)
```

Figure 4.24 compares predicted and measured speedups on the T3D, with the compute time parameter varied from $0.05us$ to $0.5us$. The general shape of the speedup curve is not changed by the order of magnitude change in compute time, indicating that on the T3D the algorithm is not dominated by communications.

On the network of workstations, the times are shown in figure 4.25. The time is out by at most a factor of three. Figure 4.26 shows how sensitive this algorithm is to the compute step value; at $0.1us$ the speedup prediction is accurate; at $1us$ the prediction is optimistic.

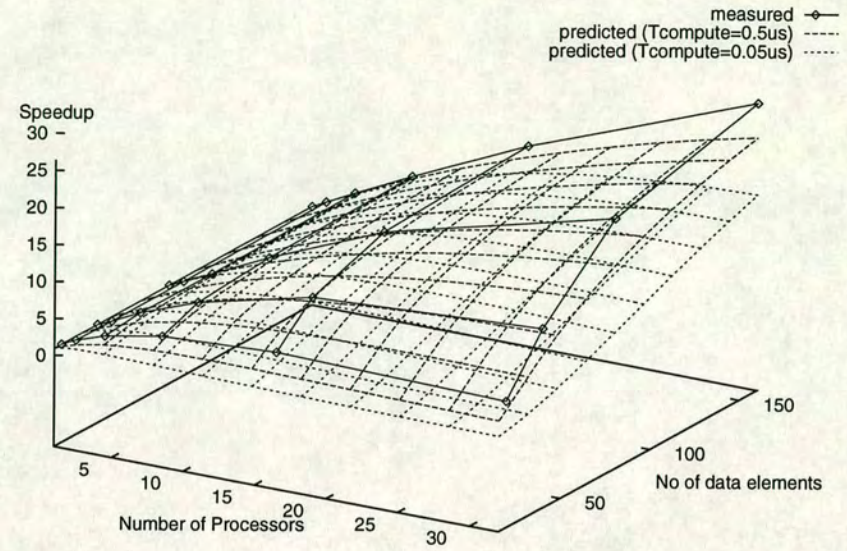


Figure 4.24: Outer measured and predicted speedups on the T3D.

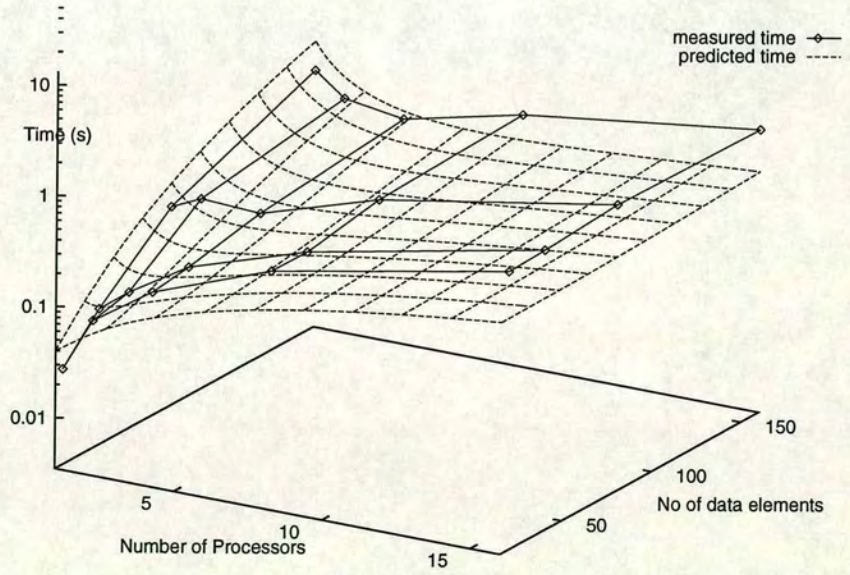


Figure 4.25: Outer measured and predicted times on a network of workstations.

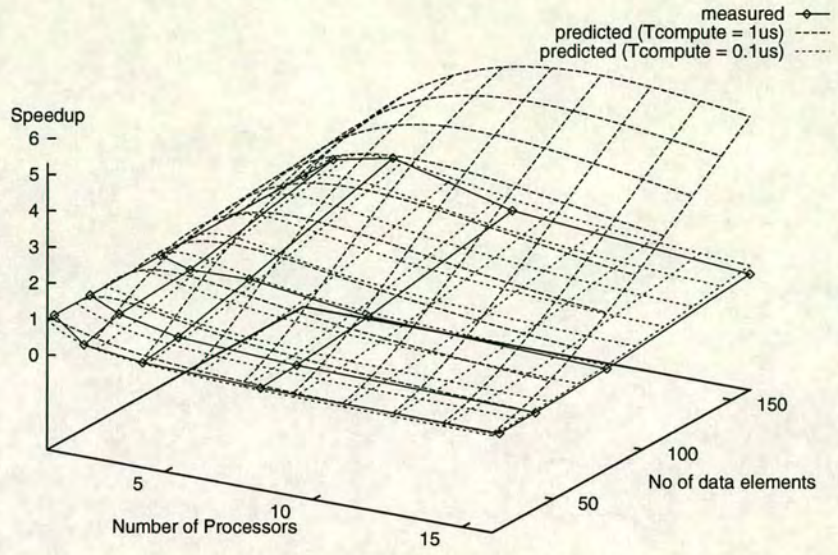


Figure 4.26: Outer measured and predicted speedups on a network of workstations.

4.3.7 Elastic net simulation (`elastic`)

This is a complex algorithm and the model reflects this. The parameters are the number of iterations and the number of cities. The total time is made up of the time required to copy all the city locations to all processes (`tmklocal`), an initialisation step `tinit` and the time for all the iterations. Each iteration involves a nearest neighbour exchange (`tneigh`), the time to compute the influence of all the cities on the net `tcity` and the time to apply the resultant force to update the net `tmove`.

```
nnet(d)          = d * 2.5
ncities(d)       = d
tmklocal(p,d)    = tallgather(p,d / p)
tinit(p,d,c)     = c*(d + nnet(d)/p)
tneigh(p,d,c)    = (nnet(d)/p)*c + 2*tsend(1) + 2*trecv(1)
tmove(p,d,c)     = c * (nnet(d) / p)
tcity(p,d,c)     = nnet(d) * c + \
       ncities(d) * ( \
              c * (d * nnet(d)/p) +\
              tallreduce(p,1) +\
              c * nnet(d)/p \
```

```
        )
titer(p,d,c)      = tneigh(p,d) + tcity(p,d) + tmove(p,d)
ttotal(p,d,c,i)   = tmklocal(p,d) + tinit(p,d) + i*titer(p,d) +\
                    tbarrier(p)
```

On the network of workstations, the times are shown in figure 4.27. The prediction is very good apart from the figures for two workstations. The measured figures for these were investigated by examining the timing diagrams. The culprit was found to be a random network delay which was delaying the inner loop communication by over 2 seconds, an occasional hazard of using a shared ethernet. The prediction is so good for all other points because the time is totally dominated by communications which are well predicted.

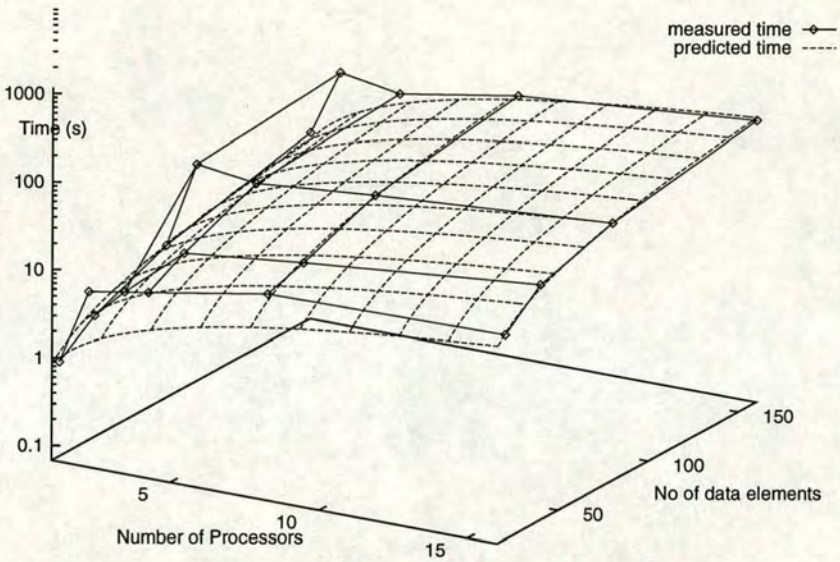


Figure 4.27: Elastic measured and predicted times on a network of workstations.

On the Cray, the initial prediction was not so good. Figure 4.28 shows that the prediction using the standard value of $0.5us$ for the compute step value led to a factor of 10 over-estimate in overall times. A value of $0.05us$ produced an accurate fit, indicating that this problem uses small enough data sizes to fit into the cache, leading to the order of magnitude better than expected performance.

4.3.8 Invasion percolation (`invperc`)

In the model, `titer` is the individual iteration time, `tenqueue` is the time needed to add a point to the local queue, `tglobalhead` is the time to determine the

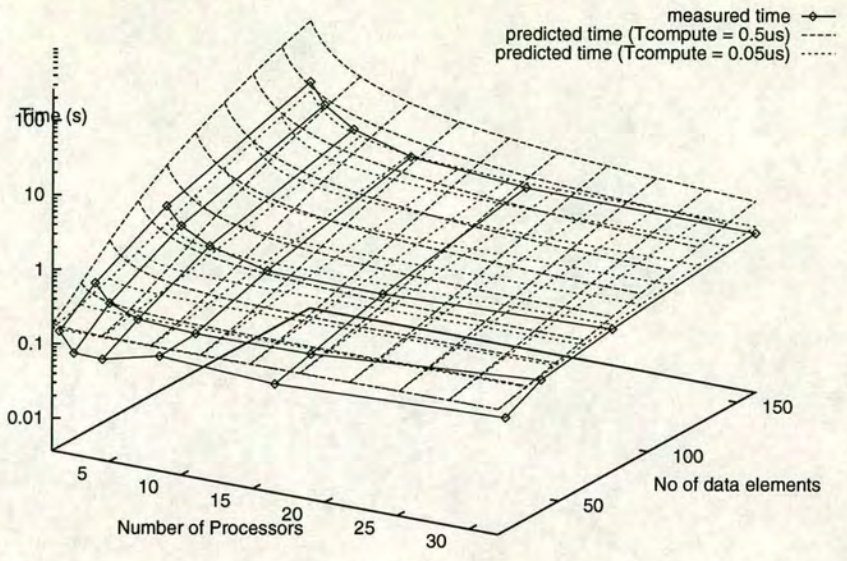


Figure 4.28: Elastic measured and predicted times on the Cray T3D.

overall head of the queue. The possible speedup comes from having a smaller queue insertion time on a parallel machine.

```
# model of invperc performance
maxqlen(d)          = d*d * fraction;
qlen(p,d)           = maxqlen(d) / (p*10);
tglobalhead(p)      = tallreduce(p,4)
tenqueue(p,d,c)     = c * qlen(p,d) * 4;
titer(p,d,c)        = tglobalhead(p) + tenqueue(p,d,c);
ttotal(p,d,c)       = titer(p,d,c) * maxqlen(d);
```

Figure 4.29 shows the measured and predicted speedups on the Cray T3D. The amount of speedup available is very sensitive to the compute step time; the initial estimate was a factor of four overly optimistic for large data sizes. This error was caused by having to guess the average queue length in the model; the times for this program are highly data dependent. Using a lower value for the compute time ($0.05us$) produced the lower bound curve in the figure.

Figure 4.30 shows the corresponding speedup curves for a network of workstations. The severe slowdown for this application is correctly predicted.

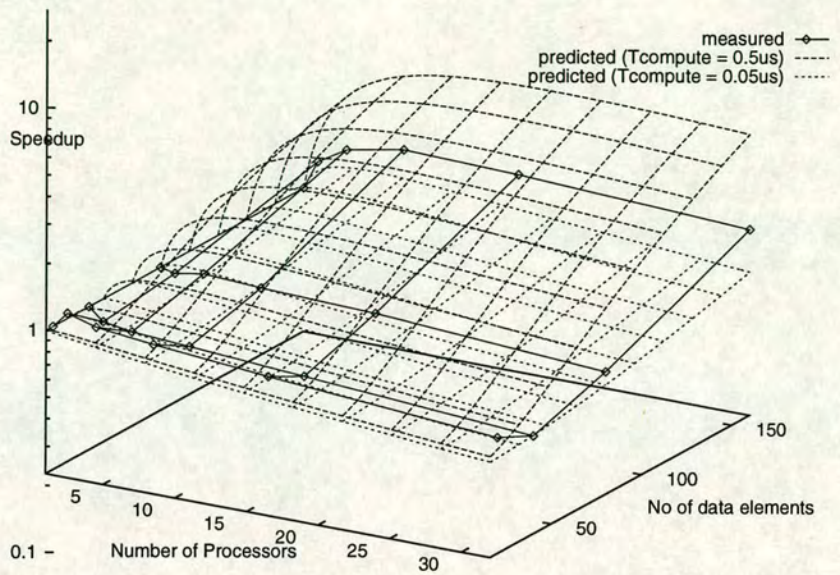


Figure 4.29: Invasion percolation measured and predicted speedups on the Cray T3D.

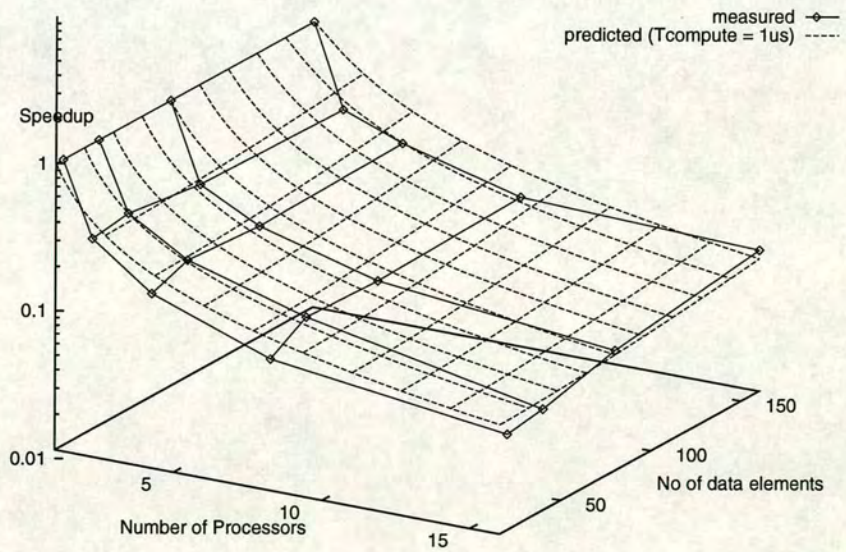


Figure 4.30: Invasion percolation measured and predicted speedups on a network of workstations.

4.3.9 Vector product (product)

The algorithm for the vector-matrix product first ensures that each process has its own copy of the entire vector, then each process may continue independently.

```
# model of product performance
tmklocal(p,d) = tallgatherv(p,(d/p))
tcalc(p,d,c)  = d * (d/p) * 2 * c
ttotal(p,d,c) = tmklocal(p,d*2.5) + tcalc(p,d*2.5,c) + tbarrier(p)
```

Figure 4.31 shows the measured and predicted times on a network of workstations. The times are dominated by communications for this problem, so adding more processors slows things down. The predictions are within a factor of two throughout; this discrepancy was tracked down to an underestimate in the `allgather` times caused by the curve fit in the MPI model. The corresponding times for the Cray T3D are shown in figure 4.32. This is a very good fit, with times dominated by computation.

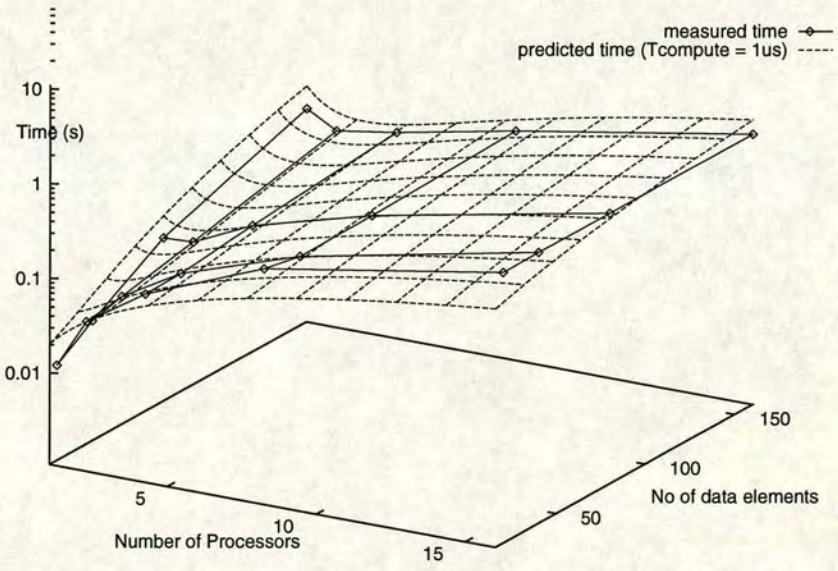


Figure 4.31: Vector product measured and predicted times on a network of workstations.

4.3.10 Successive over-relaxation (sor)

This is an iterative algorithm, so it is not possible to predict the convergence rate. It is however possible to put upper bounds on the convergence (i.e. `sormaxiters`).

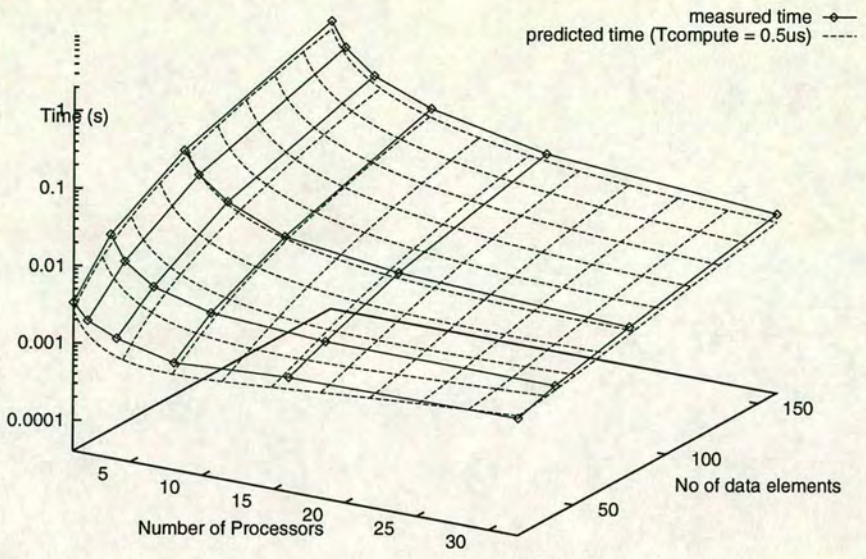


Figure 4.32: Vector product measured and predicted times on the Cray T3D.

```
# model of SOR performance
tmklocal(p,d) = tallgatherv(p,d)
titer(p,d,c)  = (d/p) * (
                        d * c * 2 +
                        c * 8) +
                tmklocal(p,d/p) +
                tallreduce(p,1)
ttotal(p,d,c) = tmklocal(p,d*2.5/p) * 2 +
                sormaxiters * titer(p,d*2.5,c)
```

Figure 4.33 shows the measured and predicted times on the Cray T3D, a very good fit. The fit for network of workstations is less good (figure 4.34) as the underlying model for collective performance is less predictable. However it does provide the useful design information that no speedup is expected.

4.3.11 Gaussian elimination (gauss)

The matrix to solve is distributed blockwise by row. The algorithm used to implement Gaussian elimination involves a single pass through the rows of the matrix in which all processes participate. The process which stores the current row then computes a pivot and broadcasts it. All processes apply this pivot row to their local section of the matrix. Figure 4.37 illustrates the performance model.

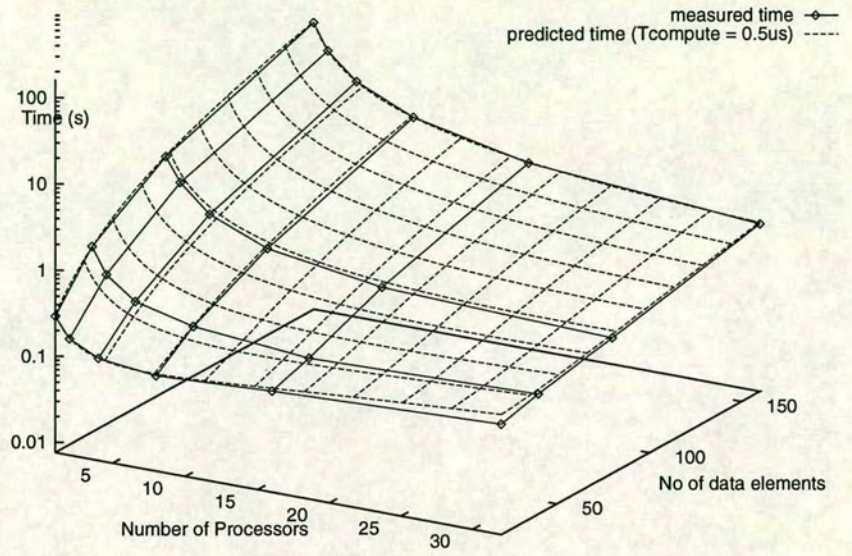


Figure 4.33: SOR measured and predicted times on the Cray T3D.

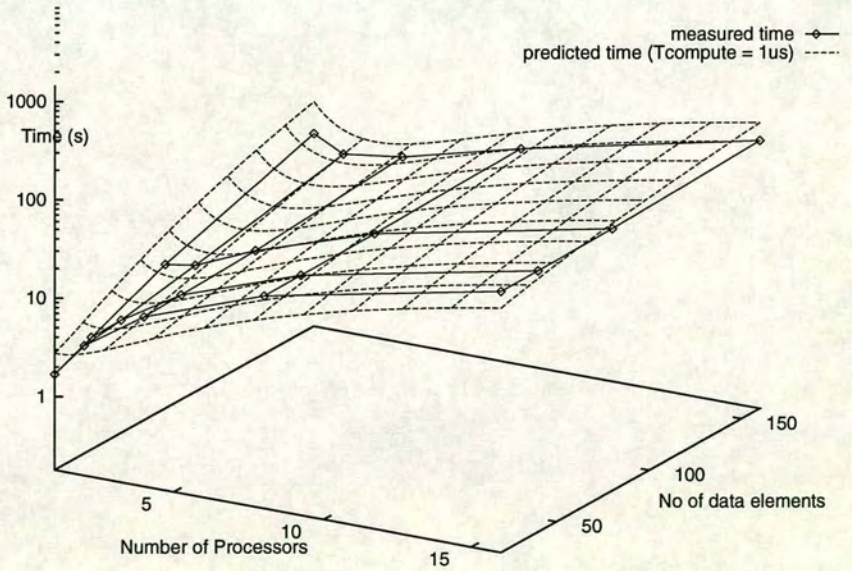


Figure 4.34: SOR measured and predicted times on a network of workstations.

```
# model of gauss performance
tcopy(d,c)              = d*c
tpivotcompute(d,c,i) = (d-i) * 2 * c + 4 * c
ttransform(p,d,c)     = d * (d/p) * c * 5
tgauss(p,d,c)          = d * (tpivotcompute(d,c,d/2) +
                              tbcast(p,d) +
                              ttransform(p,d,c) )
ttotal(p,d,c)          = tcopy(d*2.5,c) + tgauss(p,d*2.5,c)
```

Figure 4.35 shows the measured and predicted speedups on the Cray T3D. The actual speedup is worse than expected because the compute times were overestimated by a factor of five. The compute times were also overestimated on the network of workstations (figure 4.36).

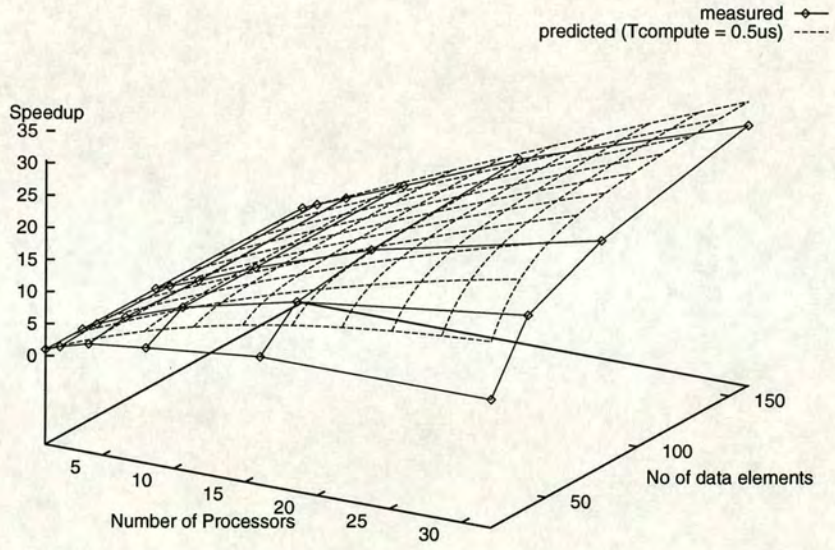


Figure 4.35: Gauss measured and predicted speedups on the Cray T3D.

4.3.12 Point normalisation (norm)

The parallel implementation of point normalisation involves a global reduction followed by independent computation phases. For design purposes it is necessary to know if the time for the global reduction swamps the total time for computation.

```
# model of norm performance
```

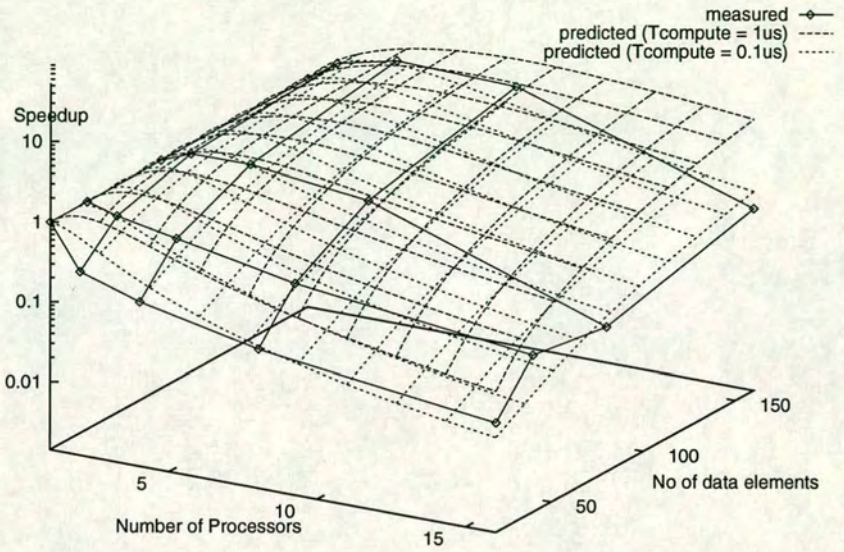


Figure 4.36: Gauss measured and predicted speedups on a network of workstations.

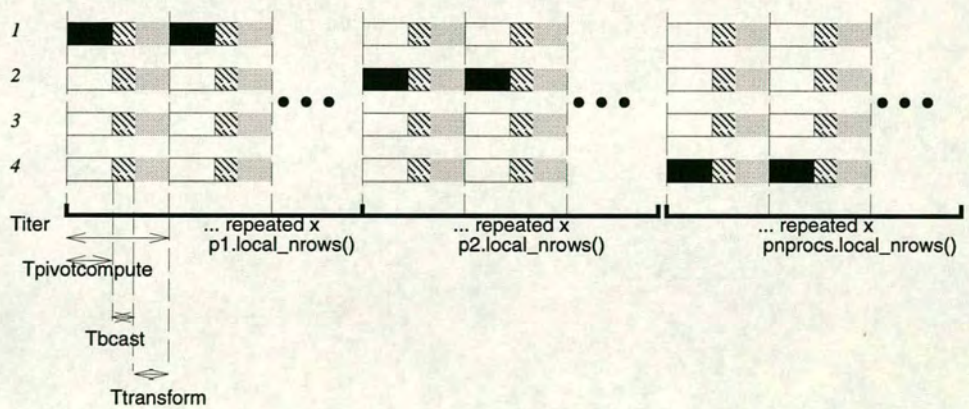


Figure 4.37: The Gauss performance model.

```
tlocalredptvector(d,c)   = d * c
tglobredptvector(p,d,c) = tlocalredptvector(d/p,c) + \
                          4*tallreduce(p,1)
tnorm(p,d,c)             = (d/p) * 8 * c
ttotal(p,d,c)            = tglobredptvector(p,d,c) +
                          tnorm(p,d,c) + tbarrier(p)
```

Figure 4.38 shows the predicted and measured times on a network of workstations. The algorithm runs more slowly as more processors are added, as predicted. However the actual times for the single processor runs are faster than predicted. This is because no communication calls are actually made, but the simple model doesn't include this special case. The prediction for the Cray T3D is a consistent factor of two pessimistic (figure 4.39), apart from the single processor case. This is because the model for the `allreduce` time (which dominates) is not accurate for the very small amounts of data used here.

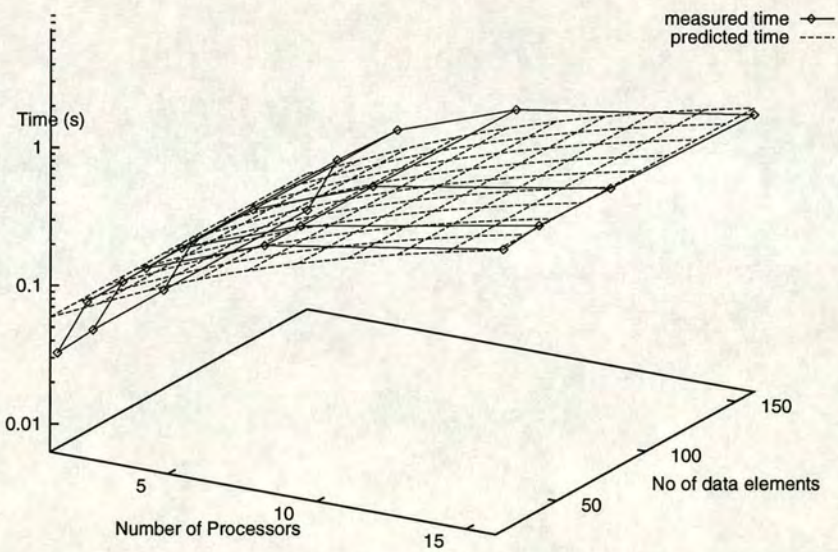


Figure 4.38: Point normalisation predicted and measured times on a network of workstations.

4.3.13 Weighted point selection (`winnow`)

```
# model of winnow performance
maxpts(p,d)          = d * d * maskpercent
minpts(p,d)          = maxpts(p,d) / p
```

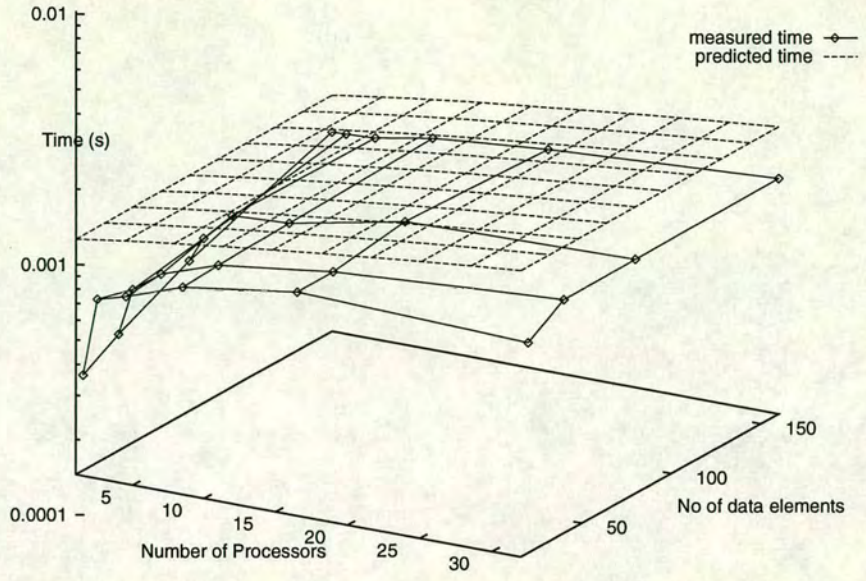


Figure 4.39: Point normalisation predicted and measured times on the Cray T3D.

```
localpoints(p,d,unevenness) = unevenness     * maxpts(p,d) +
                              (1-unevenness)* minpts(p,d)
stride = 4
tbuildlocal(p,d,c)    = c * (d/p) * maskpercent * 5
tminmaxtot(p)         = 3 * tallreduce( p, 1 )
tpivotcompute(p,d,c) = 3 * p * c +
                       localpoints(p,d,c) * 3 * tcompute
tbuffercompute(p,c)   = 2 * p * c
tlocalsort(d,c)       = d * log(d) * c *4
tpackstrided(p,d,c)   = c * 2 * d
ttotal(p,d,c,e) = tbuildlocal(p,d,c) +
                  tminmaxtot(p) +
                  tpivotcompute(p,d,c) +
                  talltoall(p,1) +
                  tbuffercompute(p,d,c) +
                  talltoallv( p, localpoints(p,d,e) )+
                  tlocalsort( localpoints(p,d,e),c ) +
                  tallgather( p,1 ) +
                  tpackstrided( p, localpoints(p) / stride,c )
```

This routine includes many phases and is too complex to make computing the time by hand a sensible proposition, so GNUplot is useful as a quick tool for "what if" calculations. The function `MPI_Alltoallv` performs the redistribution in which each process sends and receives a different amount of data to/from all the others. A parameter of "unevenness" would ideally be used to work out how long the collective redistribution operation will take. The extremes of this parameter are 0.0 (equal distribution) and 1.0 (all in one processor). A mathematical definition is given below:- We have N elements distributed amongst P processors. Ideally there would be N/P in each. Actually there are $N_i (i : 0..P-1)$.

$$\begin{aligned}
\text{Deviation } D_i &= \text{Abs}(N_i - \frac{N}{P})) \\
\sum_{i=0}^{P-1} D_i &= 0 \textit{ (for equal distribution)} \\
&= 2N(1 - 1/P) \textit{ (for all elements in one processor)} \\
\text{Unevenness} &= \frac{\sum_{i=0}^{P-1} D_i}{2N(1 - 1/P)}
\end{aligned}$$

As a simplification, the *maximum* number of elements stored in any one processor was taken as the parameter to determine the expected time of `talltoallv`.

Figure 4.40 shows the measured and predicted times on the Cray T3D. The unevenness parameter was varied from 0 to 0.1; perfect data distribution led to an over optimistic prediction and 10% unevenness was about right. 100% unevenness creates no speedup. This illustrates the sensitivity of this algorithm to the data distribution.

On the network of workstations, the algorithm produces a slowdown (figure 4.41), and this is expected even with perfect distribution.

4.3.14 Vector difference (`vecdiff`)

This program computes the maximum element by element difference between two vectors. The vectors are distributed evenly across the processors; each computes the maximum difference between the local elements, and then there is a global reduction to determine the overall maximum difference.

Thus this problem is a balance between the computation time and the reduction time. The crossover point is located where the time for the local maximum computation is equal to the time for the allreduce operation.

```
# model of vecdiff performance
```

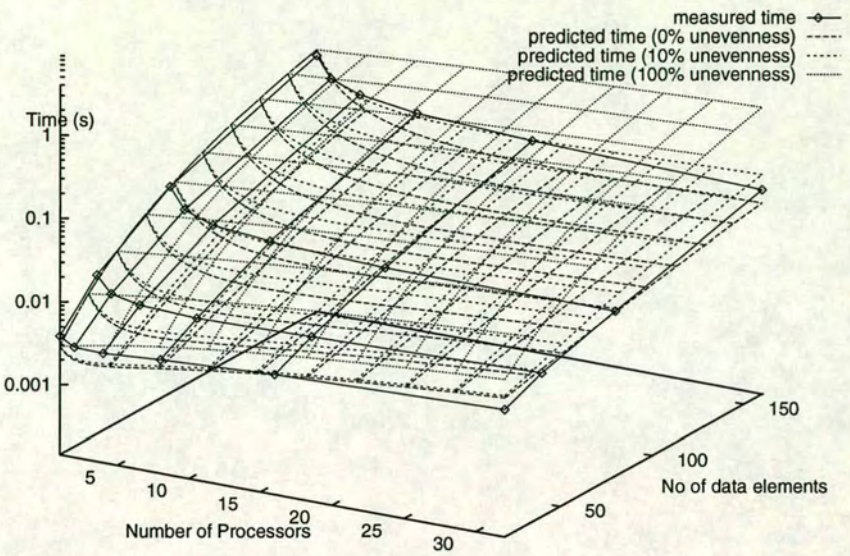


Figure 4.40: Winnow measured and predicted times on the Cray T3D.

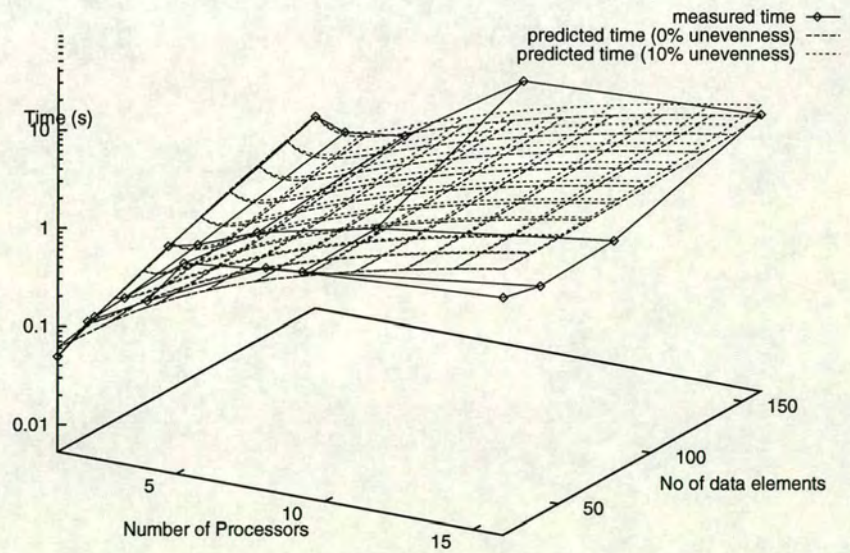


Figure 4.41: Winnow measured and predicted times on a network of workstations.

```
tlocalmax(p,d,c) = c * (d*2.5 / p)
tglobalmax(p)    = tallreduce( p, 2 )
ttotal(p,d,c)    = tlocalmax(p,d,c) + tglobalmax(p) + tbarrier(p)
```

Figure 4.42 compares measured and predicted times on a network of workstations. The times are accurately predicted for all but the single processor case, for which the communications operations take less time than the models predict. On the Cray (figure 4.43), the prediction is a factor of two pessimistic. This is because the `tallreduce` operation is a factor of two faster than the model predicts for small message sizes.

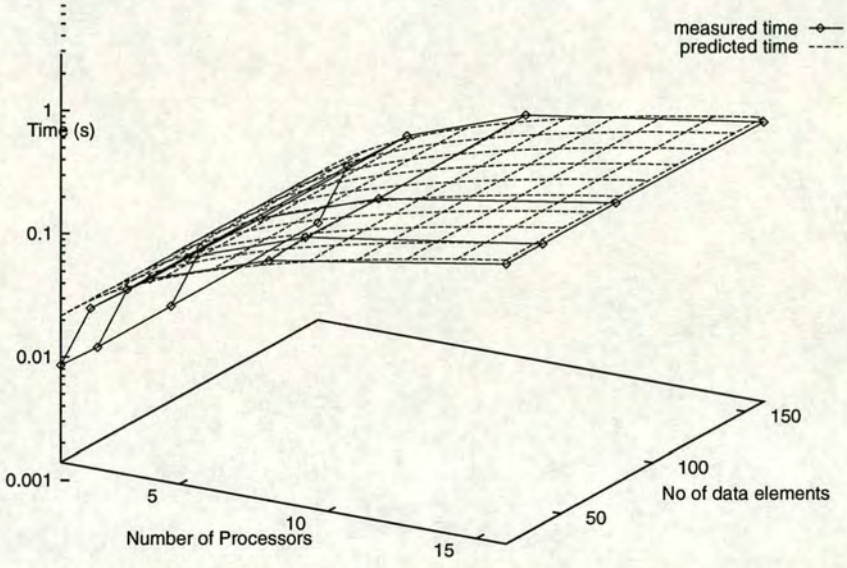


Figure 4.42: Vecdiff measured and predicted times on a network of workstations.

4.4 Conclusion

A graphing package is a very powerful and simple tool for performance prediction when used with data sheet performance models. The importance of performance modelling was highlighted by the number of slowdowns obtained with the Cowichan problems on networks of workstations.

Scalability plots are straightforward to generate, as are plots varying parameters across a wide range to check that the design performs as intended.

The best predictions were for the programs dominated either by the communications or by the computation. Programs spending equal times computing and communicating are very sensitive to minor changes, so are hard to predict.

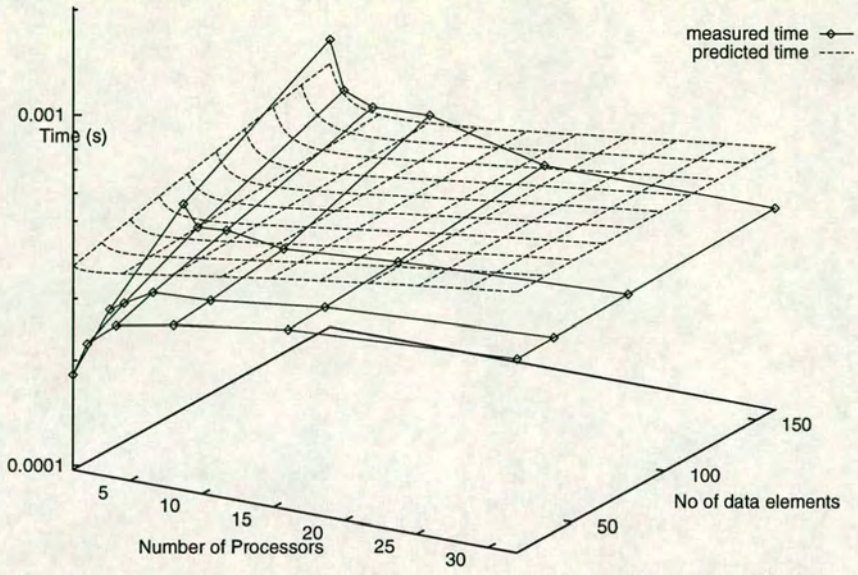


Figure 4.43: Vecdiff measured and predicted times on the Cray T3D.

The restrictions of the technique are that it requires the bounds on loop iterations to be fixed at design time and the fact that data dependencies are difficult to include. As the models for programs are generated by hand, there is a danger that seemingly unimportant phases of the algorithm will be left out, leading to overly optimistic predictions.

These restrictions are removed by the reverse profiling technique described in the next chapter.

Chapter 5

Reverse Profiling

Simple performance estimates such as those in the previous chapter are good for giving an overall picture of expected speedup for simple algorithms. However, the estimates are less valuable when processes are not all doing the same thing at the same time and when there are data dependent communications. A method for including these complexities is presented in this chapter. It is based on profiling, the well established technique for measuring how much time is spent in each part of a program.

5.1 Introduction

Reverse profiling [27] applies the MPI performance model for an architecture to a user's program to generate an estimate of the run time on that architecture. The model is generated automatically by the routines described in chapter 3. It uses the MPI profiling interface to intercept the user's calls to MPI functions and to calculate the expected delay before returning control to the MPI routine to do the actual work. In this way it is possible to estimate the expected run time of a program on any architecture using a workstation as the development platform. For example, a Cray T3D model may be used on MPI on a single workstation (or vice-versa). The technique is a simplified form of discrete event simulation. It provides quick results in the majority of cases (and will perform most of the calculations for the more complex ones).

Reverse profiling is easier to use than simulation techniques (to which it bears a resemblance), but it may only be applied to a subset of all parallel programs. The big advantage is that it may be applied simply by linking with the standard profiling library of the message passing interface. It involves each process keeping track of its own simulation time and updating it whenever an MPI function is called. This means a normal trace file can be generated. A model of any machine

may be used, and any MPI implementation can be used as the development environment.

Because it does not involve full simulation, it may not be gainfully applied to non-deterministic routines, for example those employing dynamic load balancing. However, the performance model will provide the key design data for such routines (such as the minimum and maximum message times). For non-deterministic programs the method must be combined with pencil and paper calculations, or with times measured from the target machine. Non-deterministic programs are likely to strain simulators and profilers too, since a minor miscalculation of delay may affect the outcome. A large proportion of useful parallel programs are deterministic. Reverse profiling is a simple usable technique aimed at the majority of programs.

Running a reverse profiled MPI program produces a trace file which may be displayed as a timing diagram. Repeated runs may be used to produce graphs showing how performance varies with the problem size and number of processors in the machine. The machine model is supplied at run time as an environment variable pointing to a file produced by the MPI characterisation routines.

Section 5.2 describes the technique in detail; section 5.3 describes various ways of estimating computation delays, section 5.4 presents results obtained from applying reverse profiling to some of the Cowichan problems and section 5.5 concludes.

5.2 The technique in detail

The MPI interface provides a simple profiling interface; all the `MPI_` functions are also accessable with the prefix `PMPI_`. Profiling (or reverse profiling) code may be added by writing substitute `MPI_` functions which perform the necessary (reverse) profiling task and call the `PMPI_` function to do the actual work. The linker ensures that the appropriate functions are called. The compilation commands to compile a normal MPI program, to compile with a profiler and to compile with the reverse profiler are:-

```
cc prog.c -lmpi
cc prog.c -lprof -lpmpi -lmpi
cc prog.c -lrevprof -lpmpi -lmpi
```

Each process has a variable (named `the_time`) to store its current simulation time. The profiled versions of the MPI functions update `the_time` according to the performance equation for that function and write lines to the trace file.

For point-to-point communications the receiver needs to know the time the sender started sending the message in order to work out when it should arrive (figure 5.1). The minimum delay at the receiving end occurs when the message has been posted by the sender well in advance and the message has only to be copied from a system buffer. If the `send` starts at the same time as the `recv`, the receiver will suffer an additional wait time for the message to arrive. This will be worse if the sender starts after the receive does.

These delays may be estimated from several measured parameters (determined by the routines described in chapter 3):

T_{send} Time for `MPI_Send` to complete.

T_{recv} Time for `MPI_Recv` to complete if started at the same time as the corresponding `MPI_Send`.

$T_{recvmin}$ Minimum time for `MPI_Recv` if the message has already arrived.

The total delay at the receiver is given by:

$$T_{recvmin} + T_{wait}$$

$$T_{recvend} = max(T_{recvstart} + T_{recvmin}, T_{sendstart} + T_{recv})$$

so

$$T_{wait} = T_{recvend} - T_{recvstart} - T_{recvmin}$$

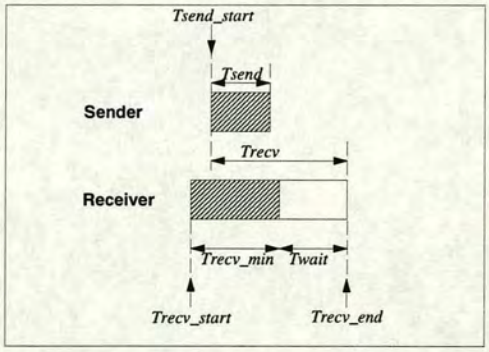


Figure 5.1: Times involved in point-point delay calculation.

For collective operations involving synchronisation (i.e. the majority of them), each process must know the start time of every other.

Thus a point-point reverse profile function looks like:

```
int MPI_Send( data, dest, ...)
{
  // Send the_time to the destination
  PMPI_Send(the_time, dest, ...);
  the_time += /* computed delay for the message */;

  // Perform the actual send
  PMPI_Send( data, dest, ... );
}

int MPI_Recv( ... )
{
  // Recv the sender's start time
  // Compute the recv delay the_time
  // function of ( the_time, sender_start, message size )
}
```

and a collective operation:–

```
int MPI_Barrier()
{
  // MPI_Allgather to get each process's the_time
  // Set local the_time to the latest of all the_times
  // Plus the computed delay for the barrier.
}
```

This works as long as two conditions are met:

1. MPI_Recv is not allowed wildcarded receives. This is because there are two receives (one for the sender time, one for the actual data) which couldn't be guaranteed to come from the same source. This problem is related to the non-determinism issue raised earlier.

2. Collective operations imply synchronisation.

A trace file is generated which may be displayed with a timing diagram tool. Each process generates a separate trace file (`p0.trace, p1.trace, etc`). Repeated runs may be combined to produce scalability graphs.

5.3 Estimating the computation delays

The reverse profiling technique has so far accounted for the communication costs quite happily, but the times for user code have not been accounted for. Even without considering compute times, useful results may be obtained since the amount of time spent in idle "wait" states can be measured from the timing diagram and the communications structure of the code is clearly visible. None of the techniques thus far encountered for estimating computation times are entirely satisfactory.

Several options for including computation are:

1. Fix it at 0. This is the dual of the PRAM model which sets the computation cost at 1 and makes communication cost 0.

2. Let the user estimate it (in units of seconds, or number of memory accesses, arithmetic operations, etc.)

3. Cycle count the assembly code.

4. Measure the times on the fly.

5. Measure the important times with a profiler off line.

Arguments may be made for all the above approaches. They make different tradeoffs between accuracy and ease of use.

Option 1, ignoring computation altogether, yields graphs showing the total communication time for an algorithm on a machine, which may be useful in itself as it shows how computation time must scale in order to make use of the machine. Option 2 is surprisingly useful. The programmer adds calls to a "`compute(N)`" macro which adds N "time steps" to the local simulated time, where a "time step" is the time taken to perform an arithmetic operation. This time is highly variable because of the influence of the memory hierarchy, but may be bracketed between likely limits (e.g. between 1 and 10 microseconds). This time step can be given as a parameter to the reverse profiler, so one may check how a design fares when given minimum expected compute step time and maximum expected communications time (the worst case for parallel algorithm scalability). Saavedra-Barrera [54] describes characterisation routines for measuring the performance of different classes of operations in Fortran and if such figures were generally available for sequential code it would make parallel design easier.

Cycle counting of assembler code (option 3) is the preferred choice of parallel machine simulators. This technique has been shown to yield very accurate time

estimates [4]. It involves an extra compilation stage, with the assembly code for the application being interpreted and augmented by a routine which inserts instructions to update a global cycle count after each basic block. Since the number of cache misses may lead to an order of magnitude variation in the execution time, a cache model is required for such simulators. This technique also requires augmented versions of all libraries used.

Experience using the Proteus `augment` tool indicated that though the technique works, it is too time consuming and awkward for quick estimates of compute time. It is also a "black box" approach and it it hard to know how reliable the estimates will be.

Option 4, measuring the compute times on the fly, is tricky on a multi-tasking system. Some multi-threading libraries provide "virtual timers" which only measure compute time consumed by the current thread, but these are not generally available. In any case, the compute times would have to be scaled for the target architecture.

The final option, profiling important subroutines on the target system and feeding the numbers back into the reverse profiler yields the most believable numbers, but requires the most effort on the part of the parallel program developer.

5.4 Results

Reverse profiling scores over the simpler technique presented in chapter 4 wherever there are data dependencies, since these are hard to incorporate into a purely analytical model. To test the suitability of reverse profiling for developing real programs, the technique was applied to a selection of the Cowichan problems. The results are presented below.

The trace output from a single run is a timing diagram showing how much time each process spends communicating and computing. This diagram is compared directly with a measured timing diagram; this is the ultimate test of a performance prediction. If a technique can generate an accurate timing diagram then it has solved the prediction problem. In practice there will be discrepencies between predicted and measured diagrams.

Comparing speedup curves is a more forgiving method of testing a prediction - even if the prediction is several orders of magnitude out in absolute terms it may produce a similar speedup curve to that measured.

The comparisons below focus on timing diagrams since they provide the ultimate check. The problem with timing diagrams is that they include a very large

amount of information, so sample extracts are presented to illustrate specific aspects of the predictions.

In the diagrams the light shade represents computation and the dark stripes are communication steps. A bar is given for each processor (`p[0]`, `p[1]`,...). The bottom bar ("`All`") shows the overall phase. On screen the different communication operations are distinguishable by different colours. The two vertical measuring lines **O** and **X** allow time intervals to be measured and displayed.

The sections below illustrate the reverse profiling process using examples from the Cowichan problems described in chapter 2.

5.4.1 Mandelbrot set generation (`mandel`)

This is an example with data dependent amounts of work to perform. In the simple technique of the previous chapter, an upper bound on the quantity of work had to be estimated (the maximum number of iterations). Reverse profiling bases timing calculations on the actual data used. This means that detailed timing diagrams can be generated, showing the expected behaviour of each processor individually, rather than the broad brush single equation for overall time used previously.

Figure 5.2 shows the predicted timing diagram produced using reverse profiling for the Mandelbrot set problem on eight SPARCstation 5 workstations connected using Ethernet, with a matrix size of $160x160$. The computation times are largest for processors 3 and 4 which compute the points inside the set, and all processors have to wait for these to complete before the problem is done. The overall time is estimated at $0.71s$ of which $71ms$ is spent in the initial barrier synchronisation, 10% of the total. Figure 5.4 shows the measured timing diagram. The measured diagram is less regular than the prediction as processors are imperfectly synchronised, leading to the staggered effect of the diagram. The total measured time is $0.23s$, a factor of 3 better than predicted, because the computation was overestimated. The initial barrier takes $78ms$, close to the expected $71ms$.

The predicted diagram for 8 processors of the Cray T3D is shown in figure 5.3. The time is expected to be determined by the maximum computation time; the initial barrier time is now insignificantly small, much less than the smallest compute time. This was confirmed by the measurement (figure 5.5), but again the absolute values of the computation time were overestimated, in this case by a factor of five.

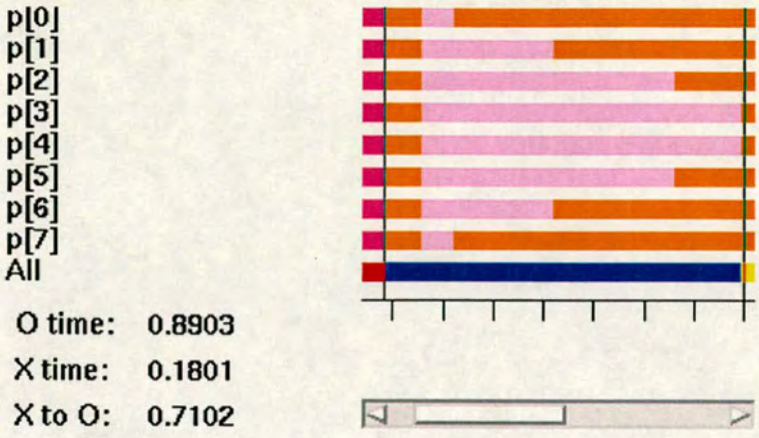


Figure 5.2: The `mandel` routine: predicted timing diagram for 8 workstations.

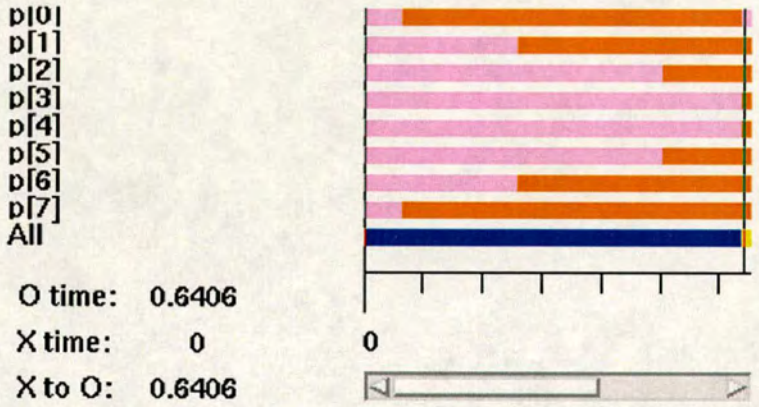


Figure 5.3: The `mandel` routine: predicted timing diagram for 8 processors on the Cray T3D.

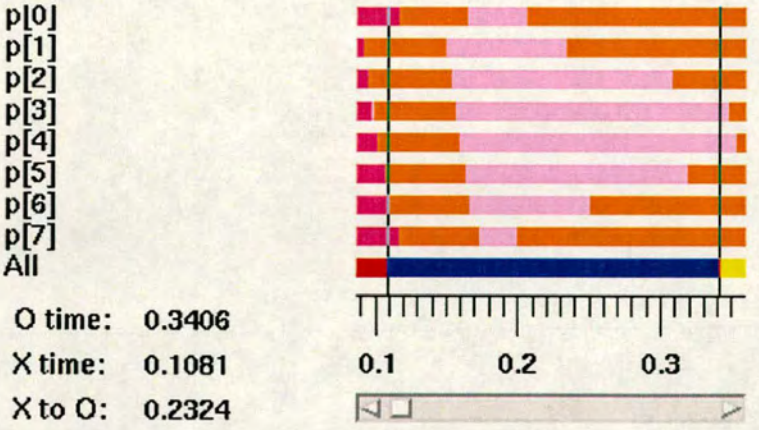


Figure 5.4: The `mandel` routine: measured timing diagram for 8 workstations.

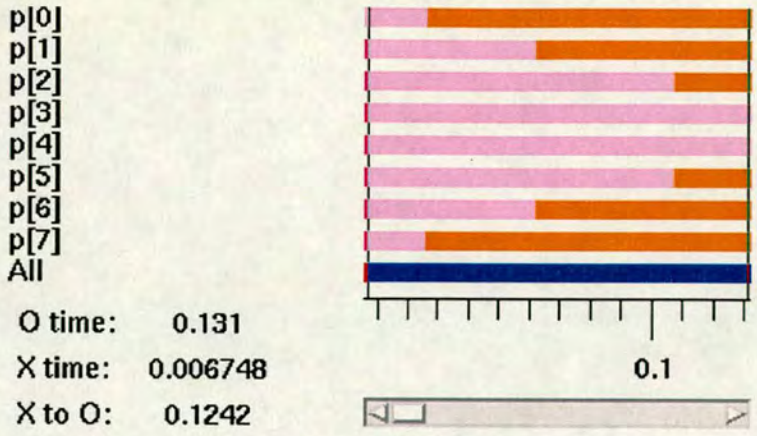


Figure 5.5: The `mandel` routine: measured timing diagram for 8 processors on the Cray T3D.

5.4.2 Elastic net simulation (`elastic`)

This example presents a quantitative comparison between timing diagrams produced by successive profiling runs, and between forward and reverse profiles. It also illustrates how I/O models may be incorporated into the reverse profile.

No two runs on a machine will take *exactly* the same time. The variation between runs on a machine provides a bound on the accuracy which can be expected from a prediction.

Table 5.1 compares the results of two runs of the `elastic` routine. The measurements were taken on 2 processors of the T3D, and the input parameter size was 128. The ratio of times varied from 0.53 for **BUSY** to 1.25 for **ALLGATHER**, a factor of 2 variation in measured times.

Phase	Prof1 (s)	Prof2 (s)	Ratio (prof2/prof1)
SEND	0.004571	0.004605	1.007365
RECV	0.367272	0.357369	0.973037
BARRIER	0.242121	0.183040	0.755984
BUSY	3.949576	2.121767	0.537214
BCAST	0.002806	0.003097	1.103726
ALLGATHER	0.004120	0.005161	1.252807
GATHER	0.249811	0.218897	0.876252
ALLREDUCE	22.954994	18.272695	0.796023

Table 5.1: Two profiles of `elastic` compared.

Table 5.2 shows the results of comparing forward and reverse profiles of `elastic`.

- SEND, BUSY, ALLREDUCE and GATHER are within a factor of 3.

- BARRIER and BCAST are out by 1000.
- ALLGATHER and RECV are out by 10.

The big errors occur where there is synchronisation. The times shown are the total times spent in each phase across all processors - so if one processor is delayed by t seconds the time for the next synchronisation operation will be extended by Nt seconds. This causes major proportional errors, particularly for synchronisation operations which execute quickly (such as the barrier). The effect of these errors in absolute terms is less significant, since the program is delayed by the t rather than the Nt.

Phase	Prof (s)	Revprof (s)	Ratio (Revprof/prof)
SEND	0.018175	0.031034	1.707511
RECV	0.165123	0.020715	0.125453
BARRIER	0.262404	0.000517	0.001971
BUSY	23.825751	41.324430	1.734444
BCAST	0.225873	0.000402	0.001780
ALLGATHER	0.003093	0.000290	0.093749
GATHER	0.000258	0.000293	1.133884
ALLREDUCE	3.274713	1.311991	0.400643

Table 5.2: Forward vs Reverse profiles of `elastic`.

To determine the causes of these errors in more detail, it is necessary to examine the timing diagrams.

Figure 5.6 shows the measured and predicted timing diagrams alongside. For this experiment, there is an overall error of 50%. Figure 5.7 shows a detail from the inner loop, which consists of long **BUSY** phases ended with an **ALLREDUCE**. The **BUSY** phases are overestimated by about 60%, and the **ALLREDUCE** is taking longer on process 1 than process 0 (whereas the prediction is for them to take the same time). Table 5.3 shows the overall times for allreduce for each processor. The explanation for this apparently major discrepancy in the allreduce prediction (taking 4 times longer than predicted on processor 1) is the variation in compute times between the two processors. Figure 5.8 shows one iteration of the inner loop. The busy time of processor 0 is $1900us$, $300us$ greater than the busy time of processor 1 ($1600us$). Because the subsequent allreduce implies a synchronisation, and because this discrepancy between the compute times of 0 and 1 was not predicted, the $300us$ is added onto processor 1's allreduce time. There is no easy way to measure the proportion of the allreduce time spent waiting for synchronisation and the proportion actually performing the allreduce (since

the synchronisation is integral to the algorithm used to implement the allreduce and not a separate operation).

Figure 5.9 shows the allreduce discrepancy in more detail. The measured time of $116us$ for the allreduce on processor 0 is close to the $102us$ predicted time; the large discrepancy is caused by processor 1 finishing its computation phase before processor 0.

Processor	Phase	Prof (s)	Revprof (s)	Ratio (Revprof/prof)
p[0]	ALLREDUCE	0.763569	0.655996	0.859118
p[1]	ALLREDUCE	2.511144	0.655996	0.261234

Table 5.3: Total times for `ALLREDUCE` for each processor.

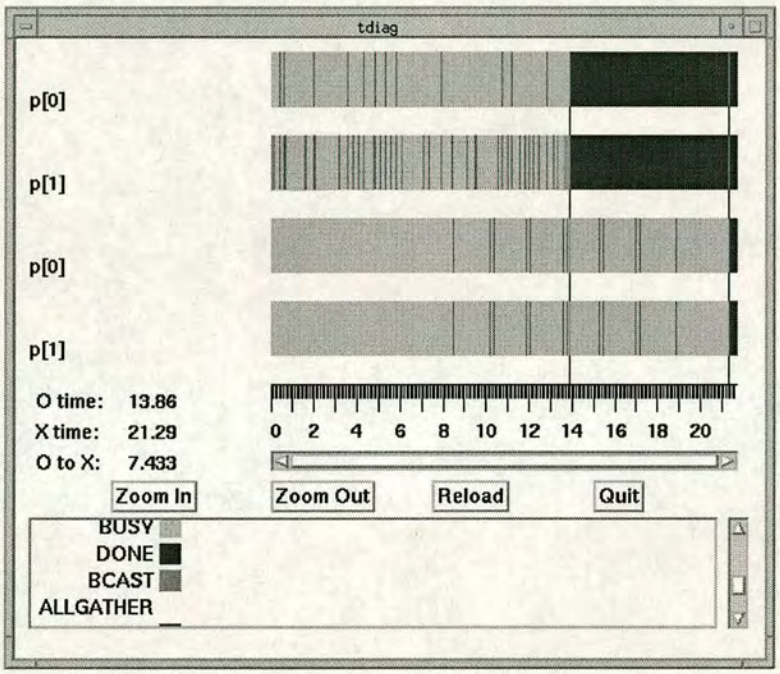


Figure 5.6: Measured (top) and predicted (bottom) timing diagrams for `elastic` - top level.

Figure 5.10 illustrates the measured time taken for the following line:

```
if (rank==0)
  printf("[%d] elastic_t iteration %d\n", net.rank, iter);
```

Output is costly, taking $3ms$ in this case. Only processor 0 performs the output, but both processors are delayed as there is a subsequent send/recv exchange of data. This appears on the diagram as an extended recv period.

This can be incorporated into the reverse profile as well - even a small amount of I/O has an effect. This may be done by changing the above line to:

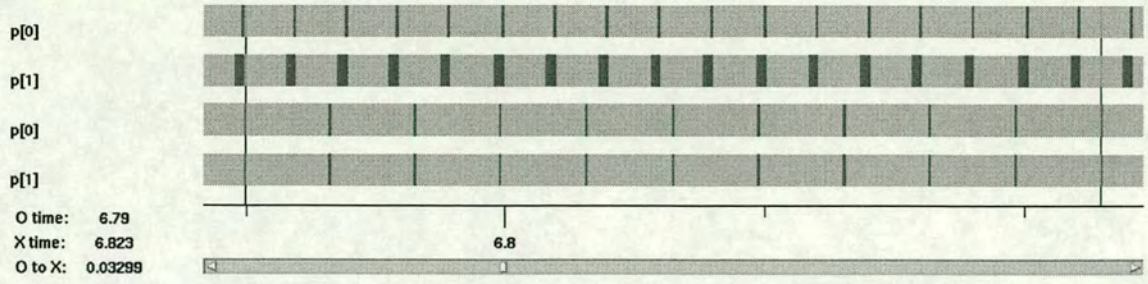


Figure 5.7: Measured (top) and predicted (bottom) timing diagrams for `elastic` - detail inside inner loop.

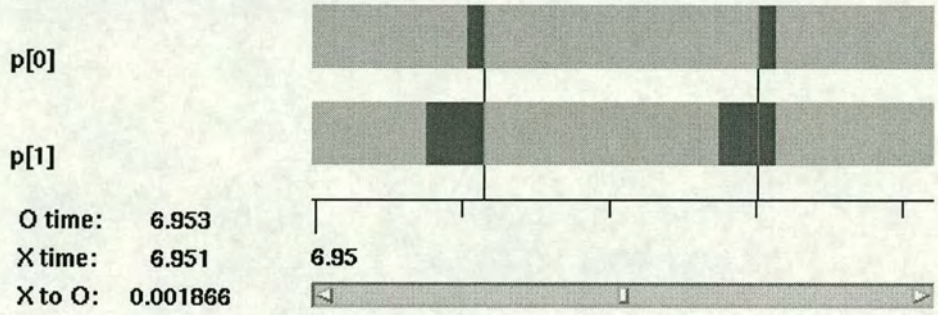


Figure 5.8: Measured timing diagram for `elastic` - one inner loop iteration.

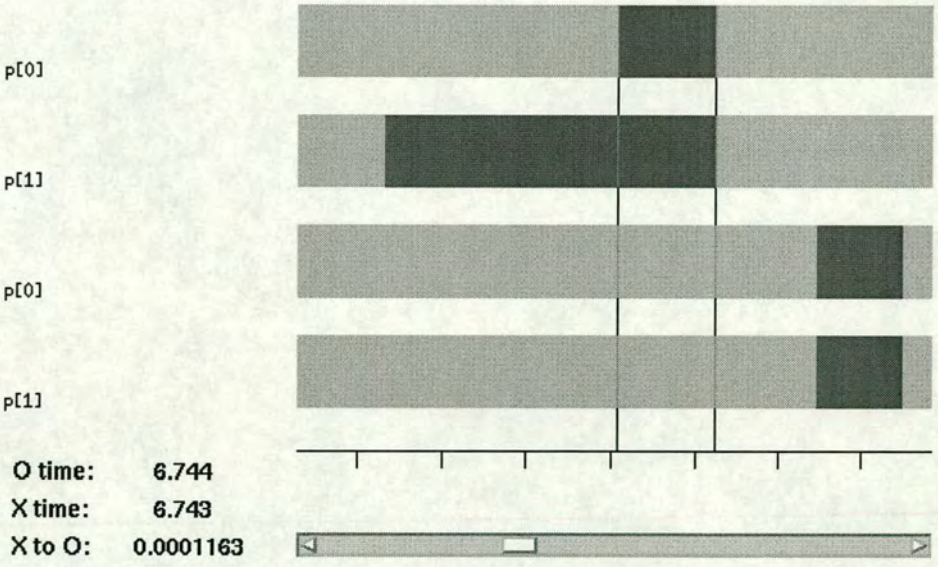


Figure 5.9: Measured (top) and predicted (bottom) timing diagrams for `elastic` - the **allreduce**.

```
if (rank==0) {
  printf("[%d] elastic_t iteration %d\n", net.rank, iter);
  compute(3000);
}
```

The equivalent extract from the timing diagram is shown in figure 5.11.

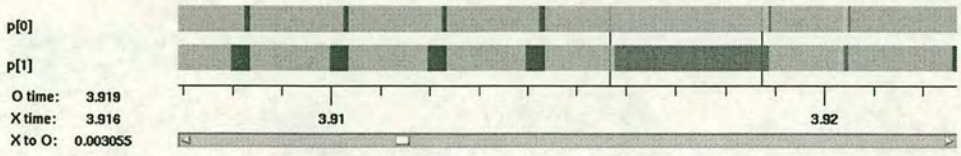


Figure 5.10: Measured time for `printf()` in each outer loop.

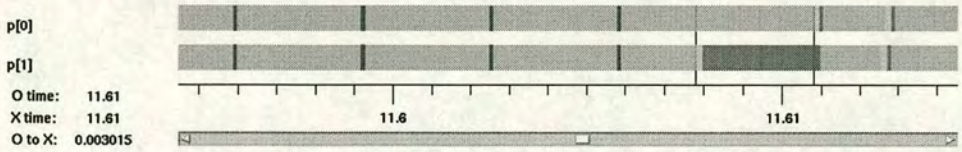


Figure 5.11: Predicted time for `printf()` in each outer loop.

Figure 5.12 shows the initial measured and predicted timing diagram for 4 processors.

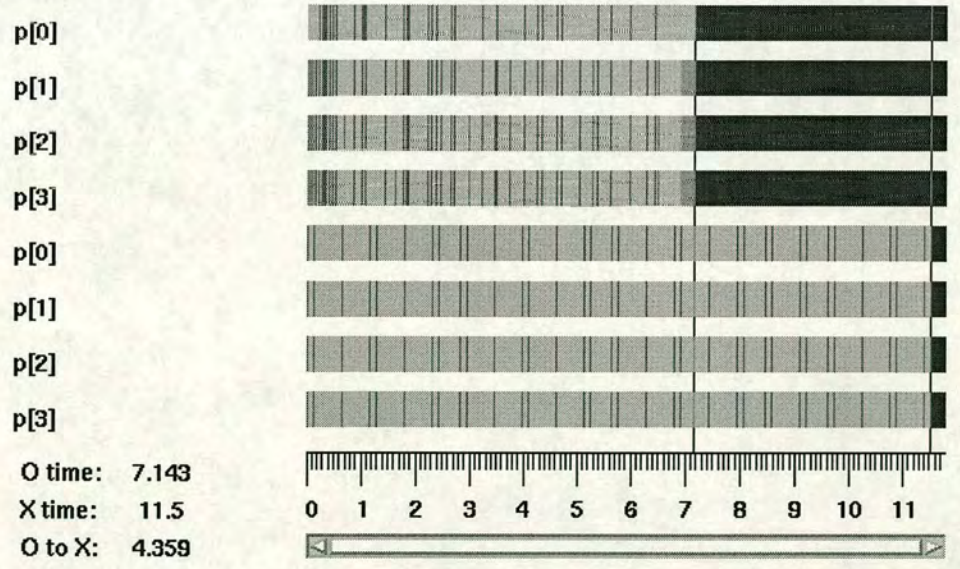


Figure 5.12: Measured and predicted times for elastic with 4 processors.

Figure 5.13 shows that the reason the predicted time is about 50% too long is that the compute phases have been overestimated; the shorter `allreduce` communications phases have been accurately predicted. The distribution of the times

actually taken for allreduce is shown in figure 5.14. This is concentrated at $150us$ with outliers scattered at higher delays. Figure 5.15 shows the distribution for busy times, with the peak at $800us$. Figure 5.16 shows the distribution of the *ratios* of predicted to measured performance; the bulk of data is clustered from 1.8 to 2 indicating that compute times are being overestimated by a factor of 2.

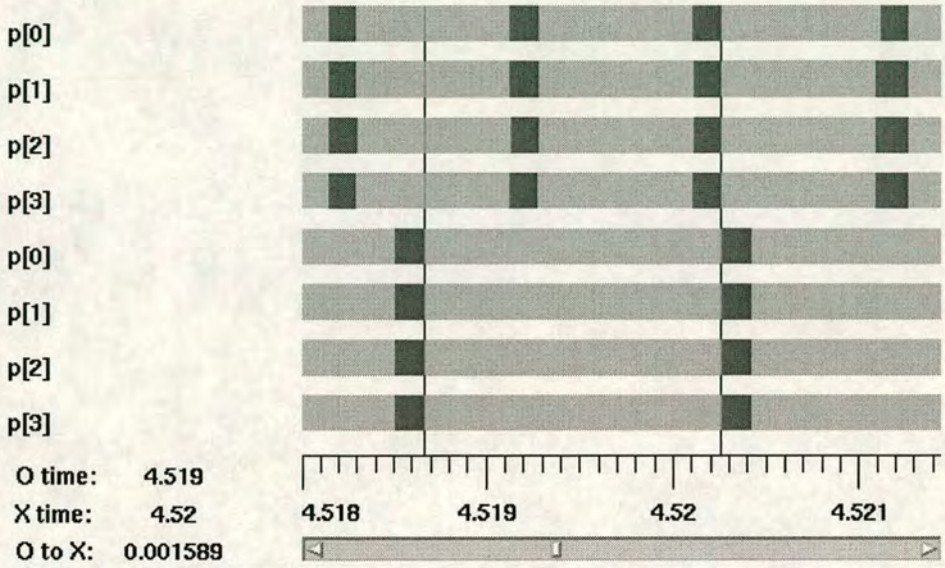


Figure 5.13: Measured and predicted times for elastic with 4 processors.

This example has shown that a prediction cannot be expected to produce a result within a factor of two, since there is a factor of two variation in successive runs. Predictions of synchronisation times are likely to be over optimistic, since prior variations across processors build up and are incorporated into the total measured synchronisation time. I/O has to be incorporated into models; even a simple `printf` reporting status takes milliseconds and could decimate available speedups.

5.4.3 Outer product (`outer`)

The example illustrated below is part of the `outer` routine. It illustrates the effect of varying the compute step time on the performance predictions and shows speedup graphs generated using reverse profiling.

`outer` is given a set of N points and computes the distance of each point from every other point. These distances are stored in a $N \times N$ matrix. Since the distance from point A to point B is the same as from B to A, the matrix is symmetric about its diagonal. For N points, $N^2/2$ distance computations are needed.

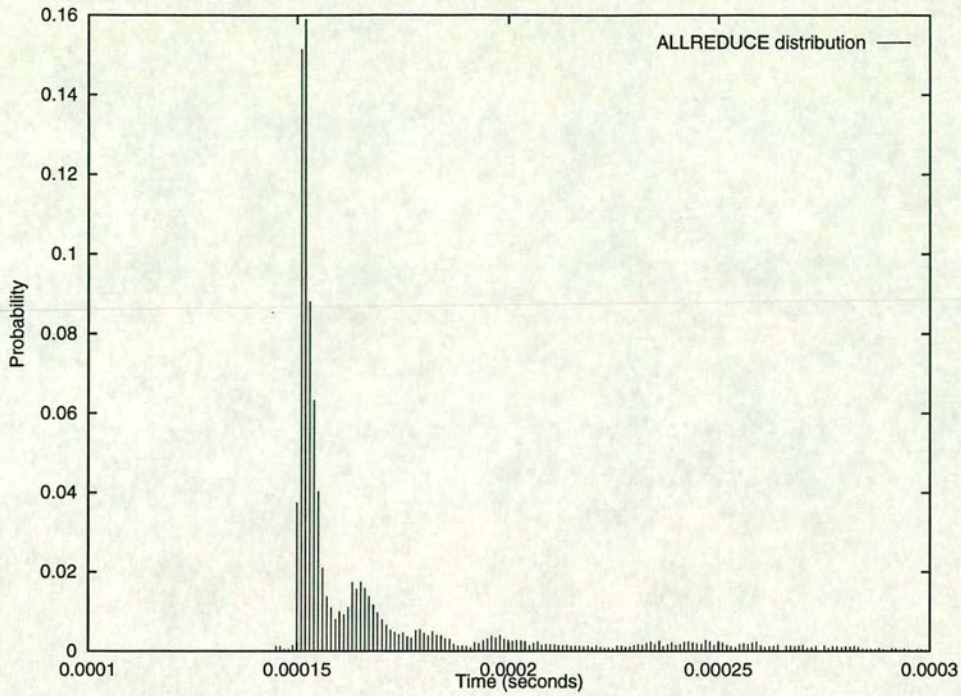


Figure 5.14: The measured distribution of `allreduce` times during elastic with 4 processors.

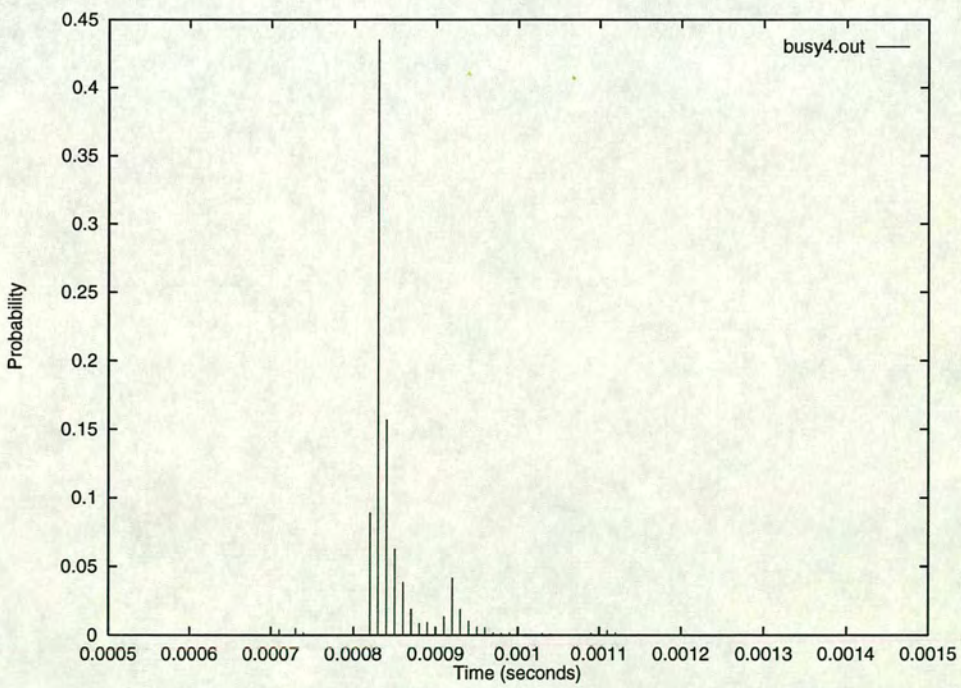


Figure 5.15: The measured distribution of compute times during elastic with 4 processors.

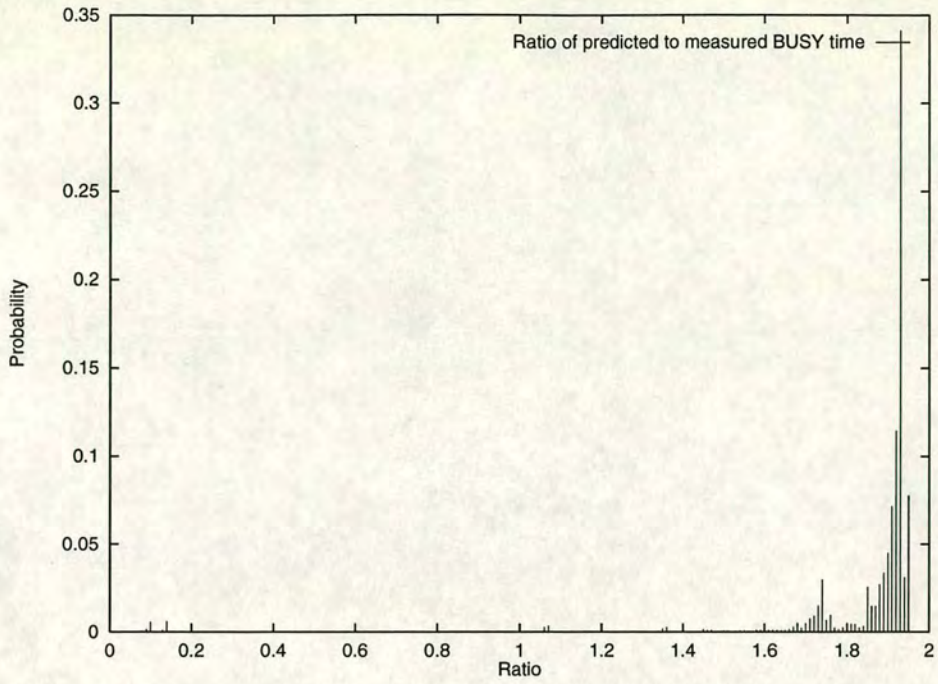


Figure 5.16: Distribution of the ratio of predicted to measured compute times.

In practice it is easier to perform N^2 distance computations than to perform half the computations and redistribute the matrix. Figure 5.17 shows the two possible distributions and figure 5.18 illustrates what the triangular redistribution would involve. Performing a triangular copy for a distributed matrix is not trivial using MPI. The MPI standard gives an elegant example of the use of datatypes to perform such copies where an entire matrix is stored on a single processor. However, coding such a copy using `MPI_Alltoallv` is not possible since `MPI_Alltoallv` requires all elements to have the same datatype. The elements in this case are irregularly shaped areas on the matrix. What is required is `MPI_Alltoallvi`, with a separate datatype constructed for each of the irregularly shaped areas. This is not part of the standard, so this approach was not taken.

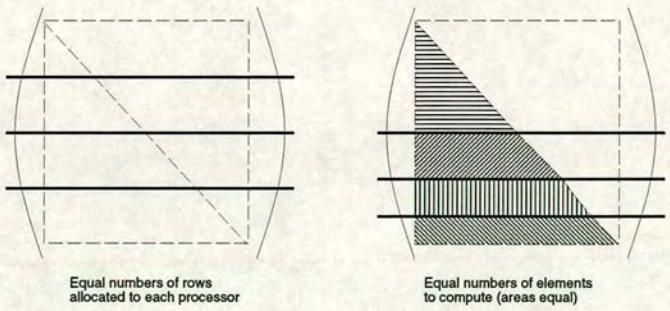


Figure 5.17: `outer` : possible distributions

Figure 5.19 shows the measured and predicted speedups, which correspond

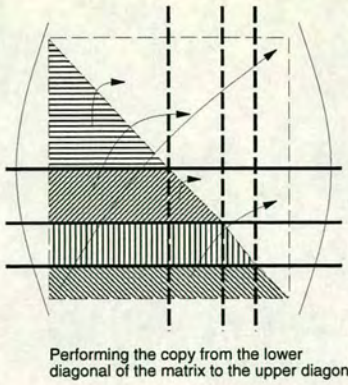


Figure 5.18: `outer` : performing the triangular matrix copy

reasonably with a compute step set between $0.1us$ and $1us$. Figure 5.20 shows that with a compute step as fast as $10ns$ the algorithm would yield minimal speedup as communication time would dominate.

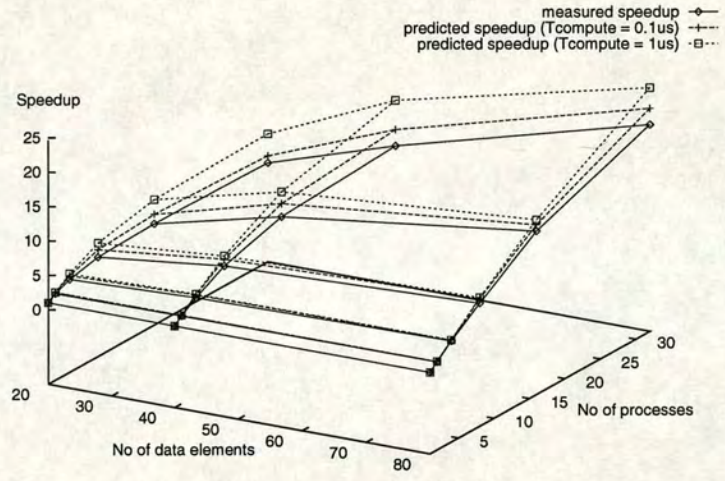


Figure 5.19: `outer` : predicted and measured speedups on the Cray T3D

5.4.4 Image threshholding (`thresh`)

Figures 5.21 and 5.22 show the predictions and measurements of the computation phase with 32 processors. The prediction is overestimated by a factor of 2 (73ms predicted, 36ms actual). It is also interesting to note that the durations of operations vary more in the measurements than the predictions. This is because collective operations are predicted to complete when all processors have completed them - but the measurements determine the exact return time (which will be different for each processor).

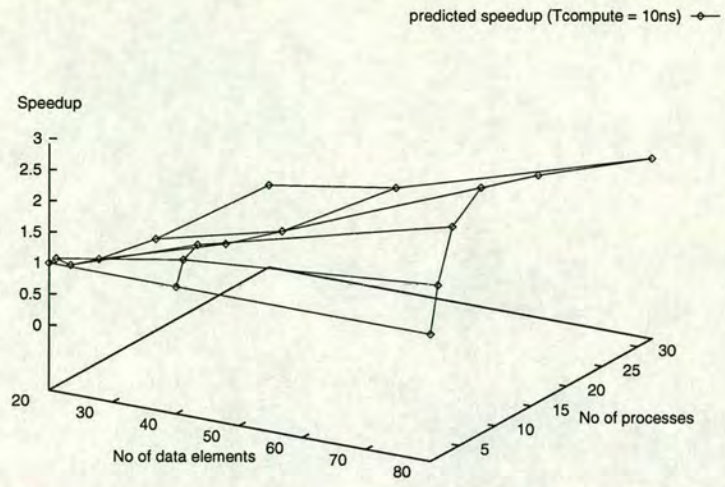


Figure 5.20: `outer` : predicted speedup with Tcompute=10ns

This simplification was made to make predictions tractable. The performance equation for `allreduce` (the collective operation which is causing the variation in this case) returns a single expected time for the allreduce across all processors. In practice, because of the algorithm used to implement the allreduce, some processors return more quickly than others, and this results in the ragged edge on the timing diagram.

The effect of this on the accuracy of predictions is not major in practice, since the discrepencies are compensated for on the next synchronisation. Those processors finishing first just have longer to wait at the next barrier.

5.4.5 The game of life (`life`)

The examples above use the *average* values to predict communications times. However the communications models of chapter 3 include more detail. This example shows how the expected minimum and maximum communications times may be used. It also illustrates the consequences of including network contention in the point to point communications models.

Figure 5.23 shows the predicted timing diagram for the game of life on two processors. Computation dwarfs the communication steps (the vertical lines on the diagram).

On 32 processors, the communication has started to make an impact (figure 5.24), taking $255us$ out of a total of $1300us$ for each life iteration (20% of the total; figures were measured from the timing diagram.) For 2 processors,

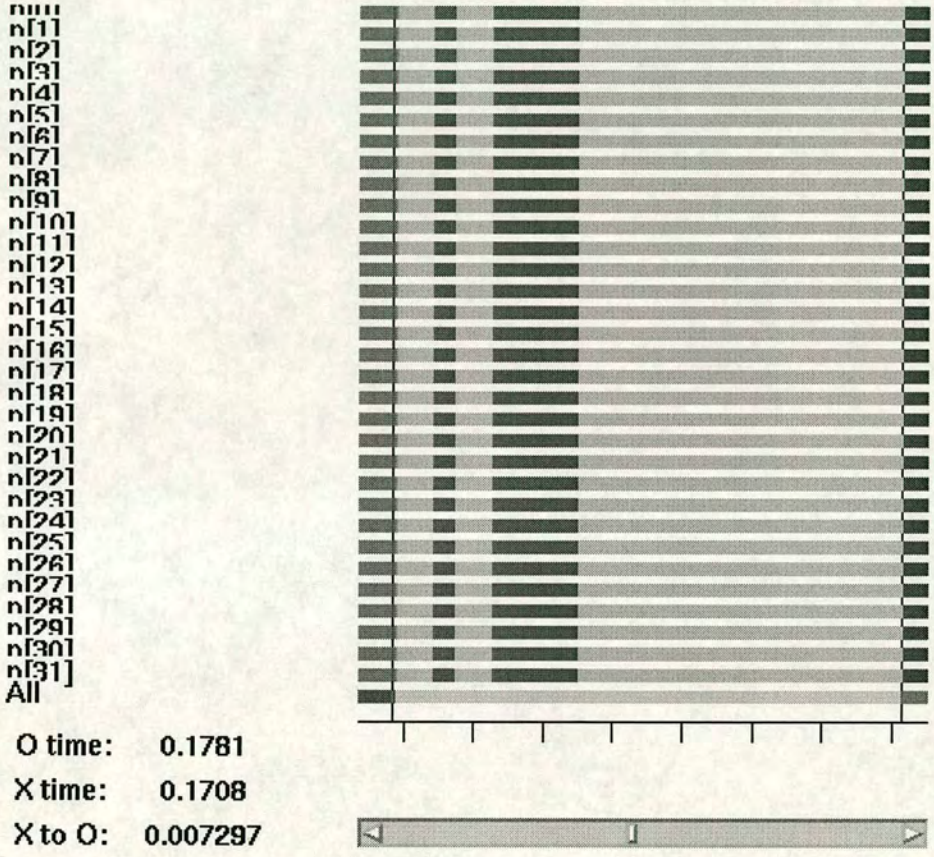


Figure 5.21: The `thresh` routine; predicted timing diagram for 32 processors.

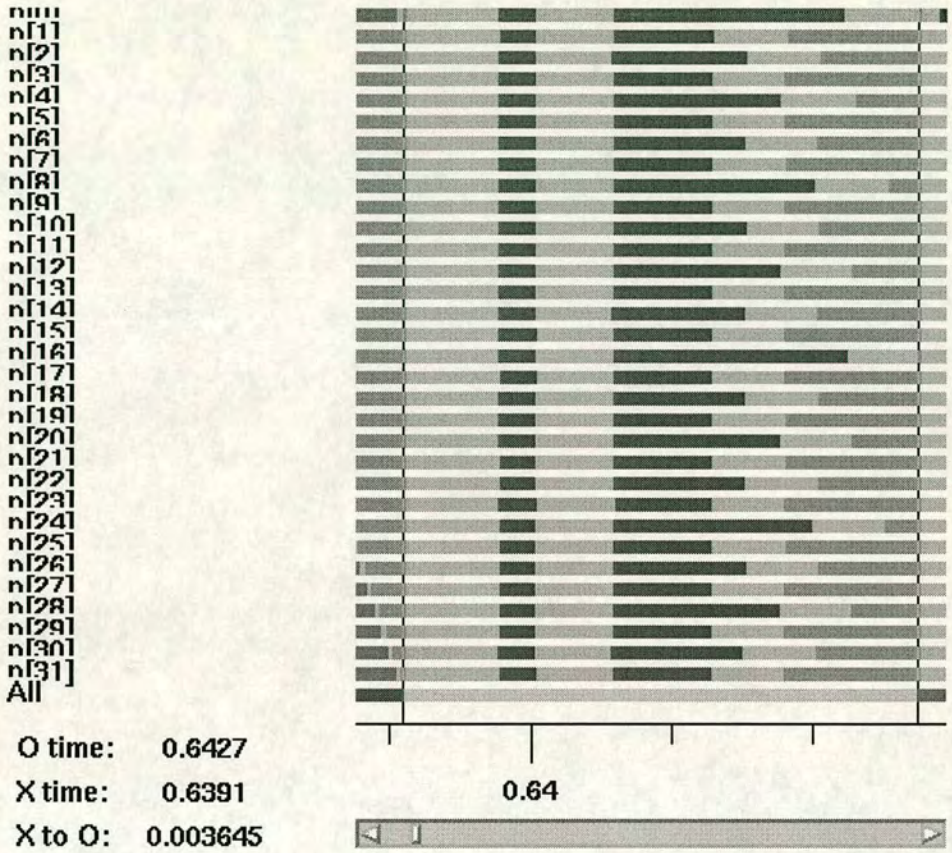


Figure 5.22: The `thresh` routine; measured timing diagram for 32 processors.

Figure 5.23: The `life` routine; predicted timing diagram for 2 processors.

communications was taking $255us$ out of a total of $16900us$ for each life iteration (1.5%).

Figure 5.24: The `life` routine; predicted timing diagram for 32 processors.

On 32 processors however, the reading and writing of files is expected to dominate, with the computation burst of activity taking only 7% of the time. Figure 5.25 shows the central computation band.

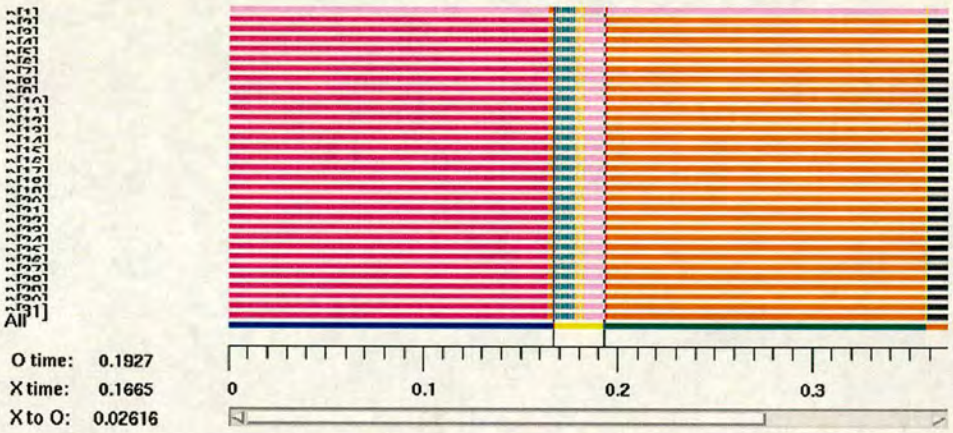


Figure 5.25: The `life` routine; predicted timing diagram for 32 processors (top level).

Comparing with measurements, figures 5.26 and 5.27 show the expected and actual times for the computation phase of life with 32 processors. Visually they are similar; the actual total time for the phase is $19.5ms$ and it was predicted at $26ms$.

Table 5.4 compares the total send and receive times spent by all processors. Send is underestimated by a factor of 2.4 and Recv is overestimated by 35%. This send error is a concern and is a result of characterising the performance of a send on an uncongested network.

Replacing the send predictions with a characterisation based on a congested network (timing the send when one half of the processors are sending to the other

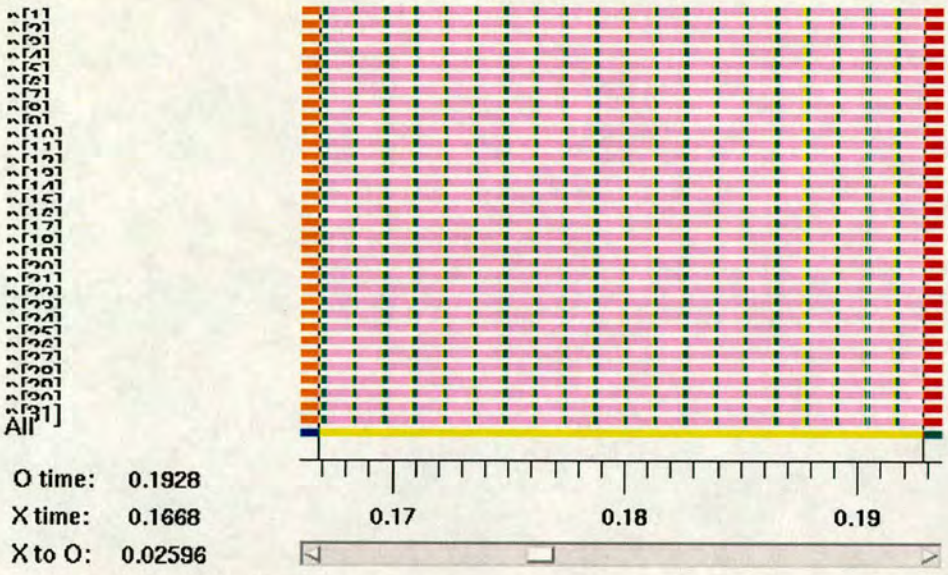


Figure 5.26: The `life` routine; predicted timing diagram for 32 processors (computation phase).

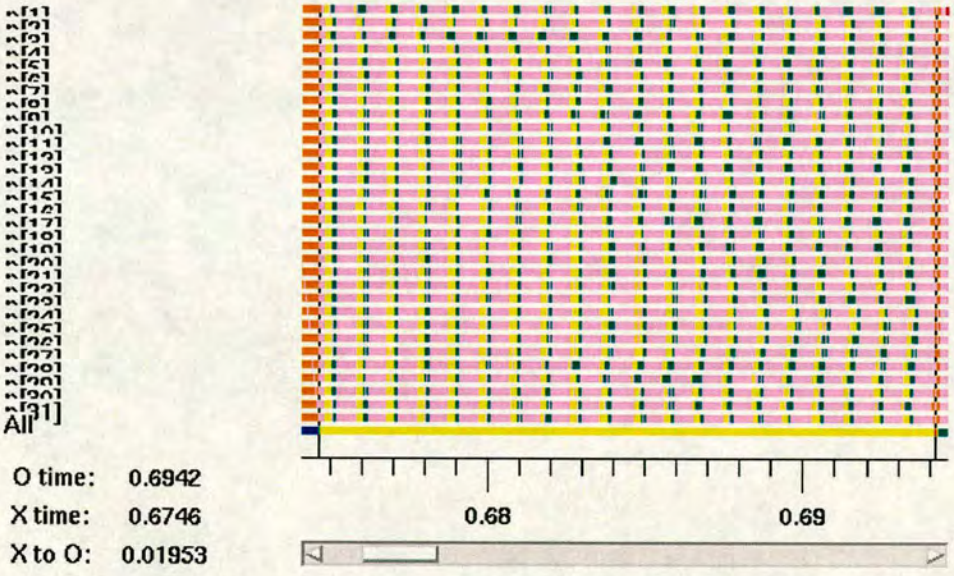


Figure 5.27: The `life` routine; measured timing diagram for 32 processors (computation phase).

Phase	Prof (s)	Revprof (s)	Ratio (Revprof/prof)
SEND	0.118	0.049	0.41
RECV	0.085	0.114	1.34

Table 5.4: Forward vs Reverse profiles of `life` : using simple send prediction.

half) yields more accurate times (table 5.5):

Phase	Prof (s)	Revprof (s)	Ratio (Revprof/prof)
SEND	0.118	0.084	0.71
RECV	0.085	0.109	1.29

Table 5.5: Forward vs Reverse profiles of `life` : using congested send prediction.

The equations for MPI performance give confidence intervals on the parameters, so predictions are possible based on both minimum and maximum expected communications times. Figure 5.28 shows the computation phase of `life` using the maximum and minimum expected communications delays. The maximum total time for the phase is $28.1ms$, the minimum $25.7ms$. Each communication step in the inner loop has a maximum expected time of $363us$ and a minimum of $244us$. Figure 5.29 shows the variation in the time for a single neighbour exchange communications step. The actual measured time of each communications step is $360us$, within the predicted range.

These results were obtained by setting the environment variable:

```
export REVPROFMODE=MIN
export REVPROFMODE=MAX
or export REVPROFMODE=AVG
```

before running the program.

Bracketing the computation time is also important; this may be done by using different values of the `tcompute` computation step time (set within the MPI characterisation file).

For example, improving computation performance by a factor of 10 leads to the profile in figure 5.30. The computation phase time is now $8.2ms$ (down from $26.9ms$ before), i.e. a factor of 3 improvement in algorithm performance can be expected from a factor of 10 improvement in processor performance. At this computation rate, communications takes 75% of the time of each life iteration.

Going the other way, slowing the computation time down by a factor of 10 yields a total computation phase time of $213ms$. Communications only occupies 3% of the time.

5.4.6 Weighted point selection (`winnow`)

The input for the test of `winnow` (weighted point selection) was the Mandelbrot set. The predicted timing diagram for the first phase of computation is shown in figure 5.31. The load is not balanced; the input Mandelbrot set has high values

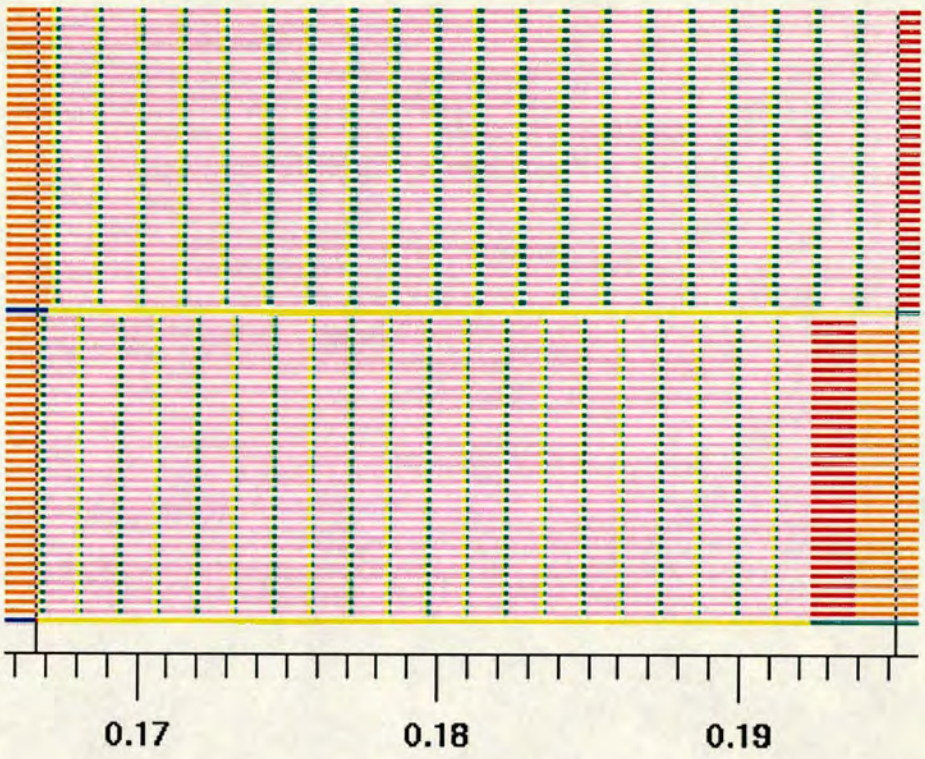


Figure 5.28: The `life` routine; predicted timing diagram for 32 processors using maximum (top) and minimum (bottom) expected communications times.

concentrated in the centre, so a disproportionate amount of computation is done by the central processors.

Figures 5.32 and 5.33 show the predicted and measured times for the top level of winnow computation. The measurement is twice as fast as the prediction. This error is due to the computation running approximately twice the expected speed; the predictions of communication times are within expected limits. The extreme imbalance of the load, with processor zero executing a very long computation step and holding up the rest of the processors is highlighted by the prediction.

5.5 Conclusions

This chapter has illustrated the application of reverse profiling with examples drawn from the Cowichan problems. Communications dominated programs are predictable to within a factor of two, and programs dominated by computation to a factor of ten. This is not as accurate as simulation techniques presented in the literature, but is sufficient to answer the question: "will this application run faster or slower on a parallel machine?".

The important features of the technique are:

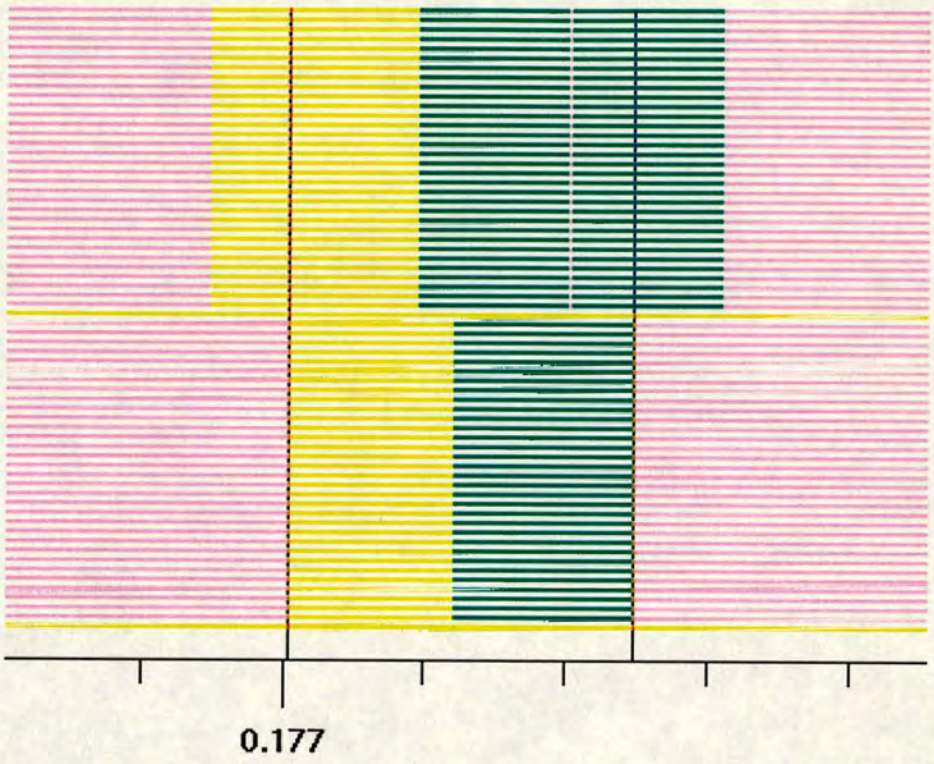


Figure 5.29: The `life` routine; predicted timing diagram for 32 processors (detail). The top half shows the maximum expected time for a communication step, the bottom half shows the minimum expected time.

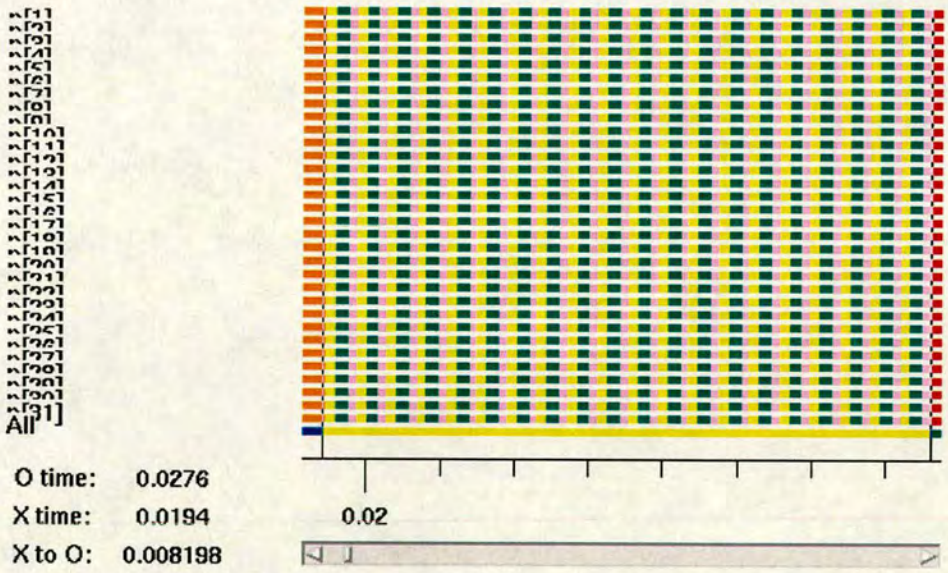


Figure 5.30: The `life` routine; predicted timing diagram for 32 processors, with computation speeded up by a factor of 10, communications left the same speed.

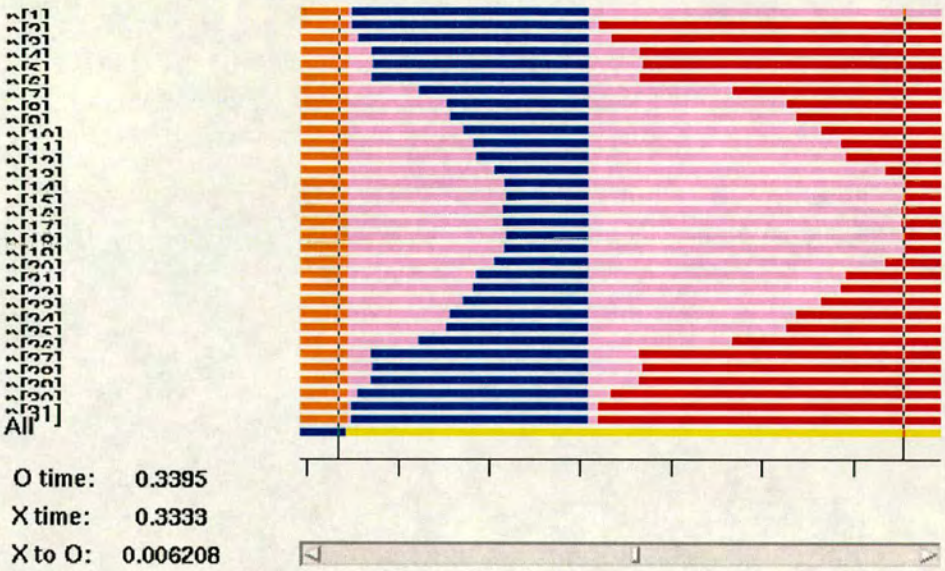


Figure 5.31: The `winnow` routine: predicted timing diagram for 32 processors. The first stages of computation have the appearance of the edge of the Mandelbrot set.

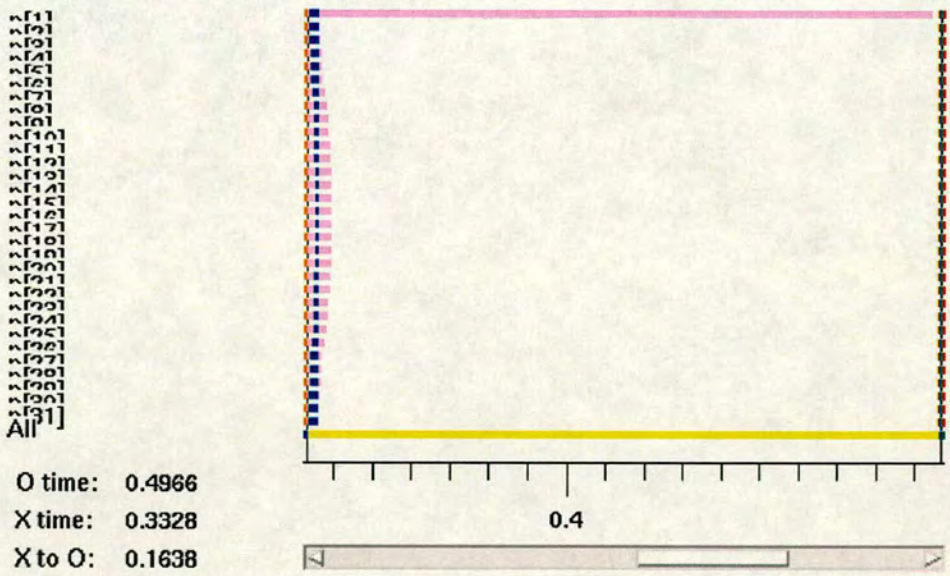


Figure 5.32: The `winnow` routine: predicted top level timing diagram for 32 processors.

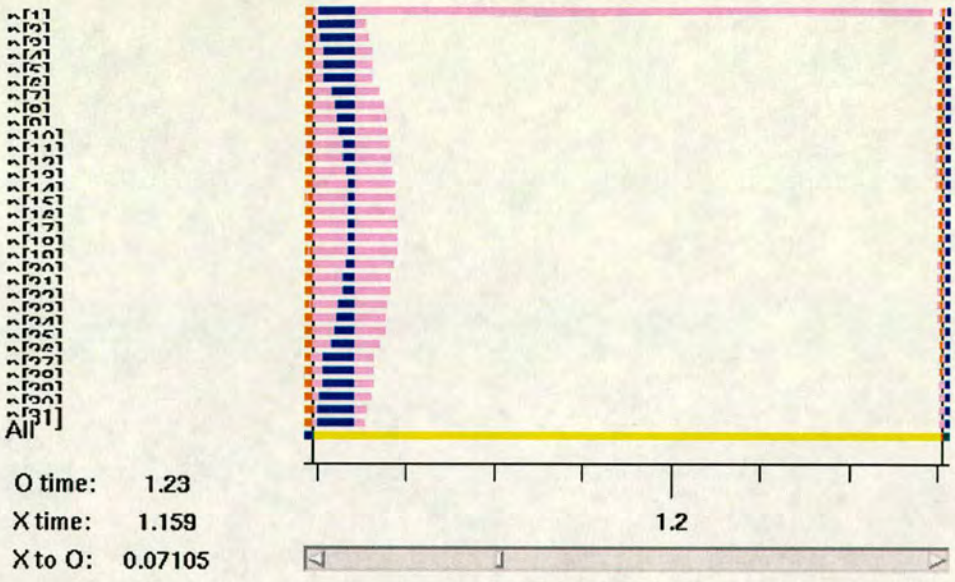


Figure 5.33: The `winnow` routine: measured top level timing diagram for 32 processors.

- It is as easy to use as normal profiling
- Predictions for any target architecture may be made on the development platform
- Predictions are based on actual data distributions
- Data dependent communications may be included
- Expected timing diagrams are produced, showing graphically where most time is spent
- Non deterministic communications are not supported

Reverse profiling offers a very quick and easy method of performance prediction for MPI programs. Unlike simulation techniques it builds directly upon the full and complete MPI libraries available now. It does not attempt to handle non-determinism but this is the area in which existing profilers and simulators produce the least believable results. It works with any MPI implementation which provides the standard profiling interface, so predictions may be performed in parallel.

The next chapter investigates the use of a discrete event simulation tool for performance prediction, to plug the non deterministic gap.

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 [14] 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 [5] 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 [51], Proteus [4] and PS [3]) 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 [31], [28] and [29].

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++ [56], 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.

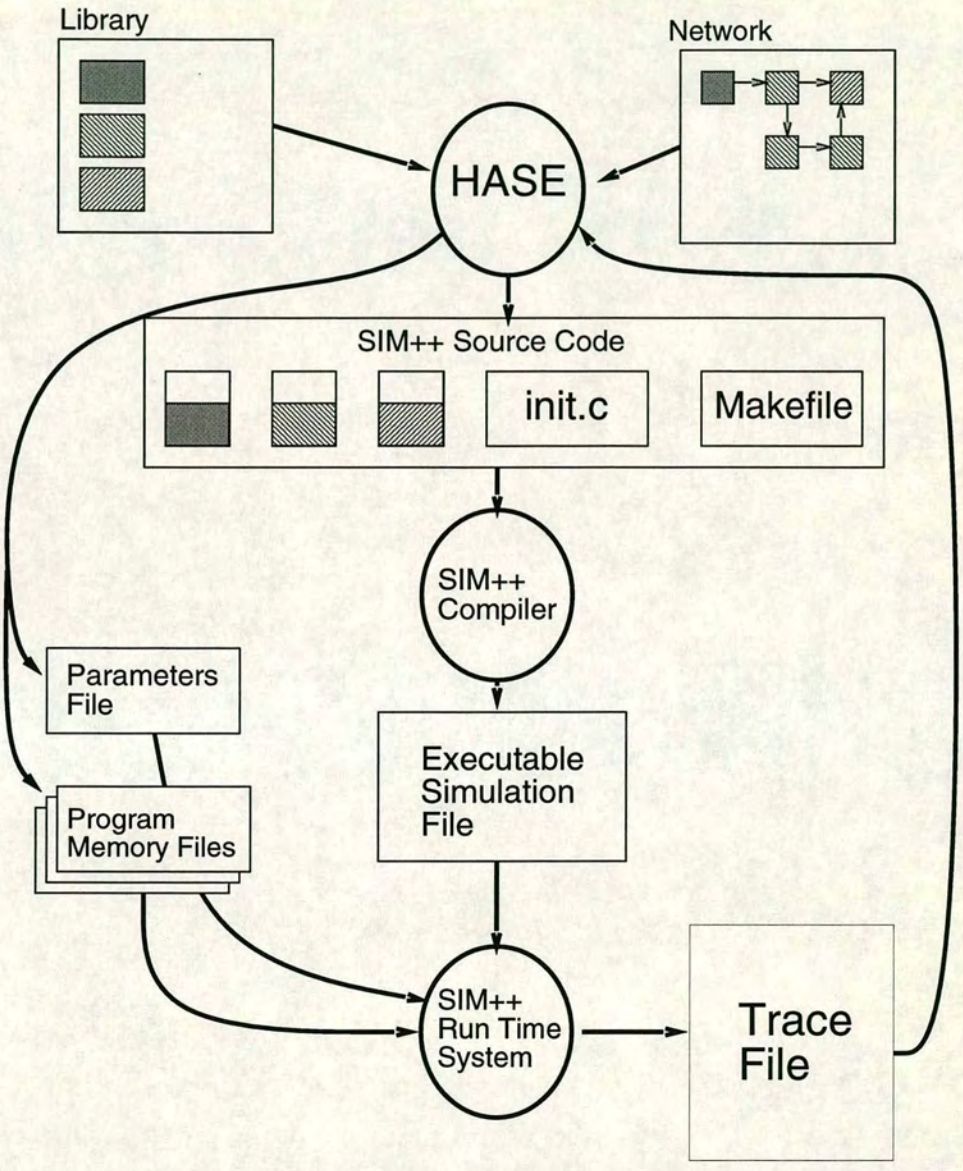


Figure 6.1: The top level design of HASE.

- **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.

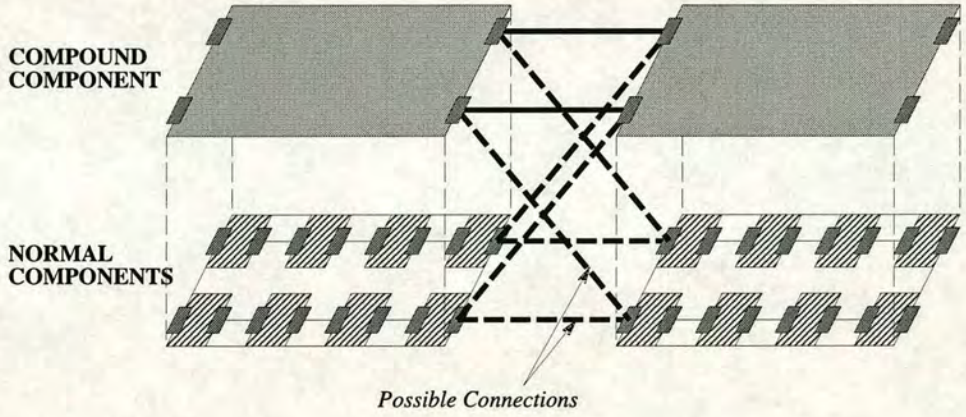


Figure 6.2: Two subdivided entities are connected by their external ports.

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

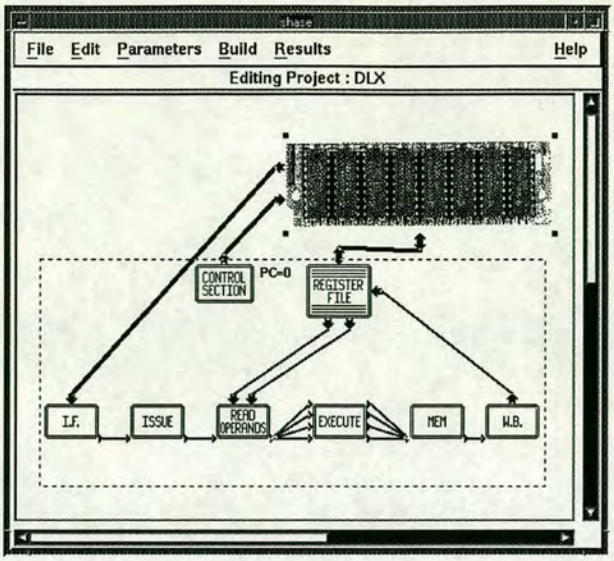


Figure 6.3: The HASE user interface.

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
```

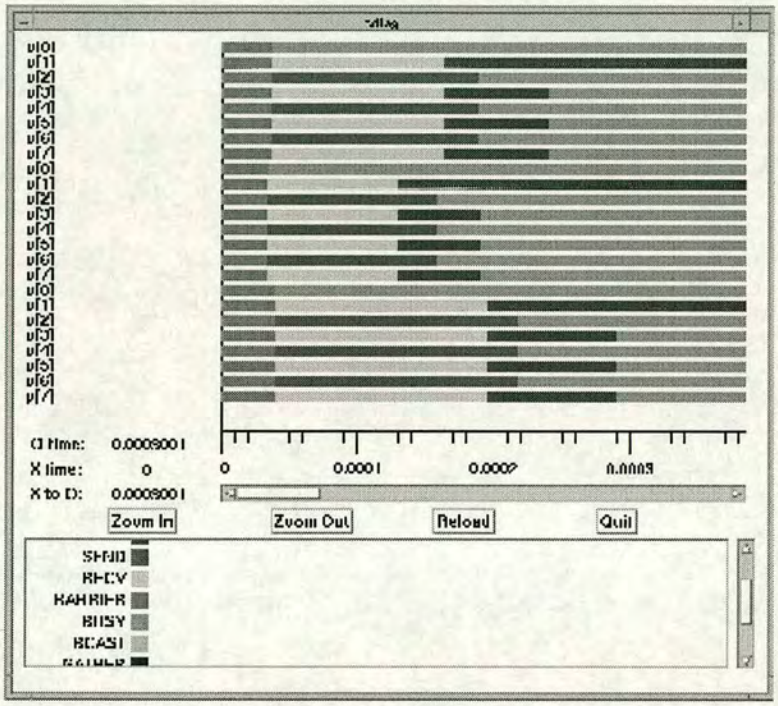


Figure 6.4: A timing diagram display.

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 [23]. 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 [32] 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).

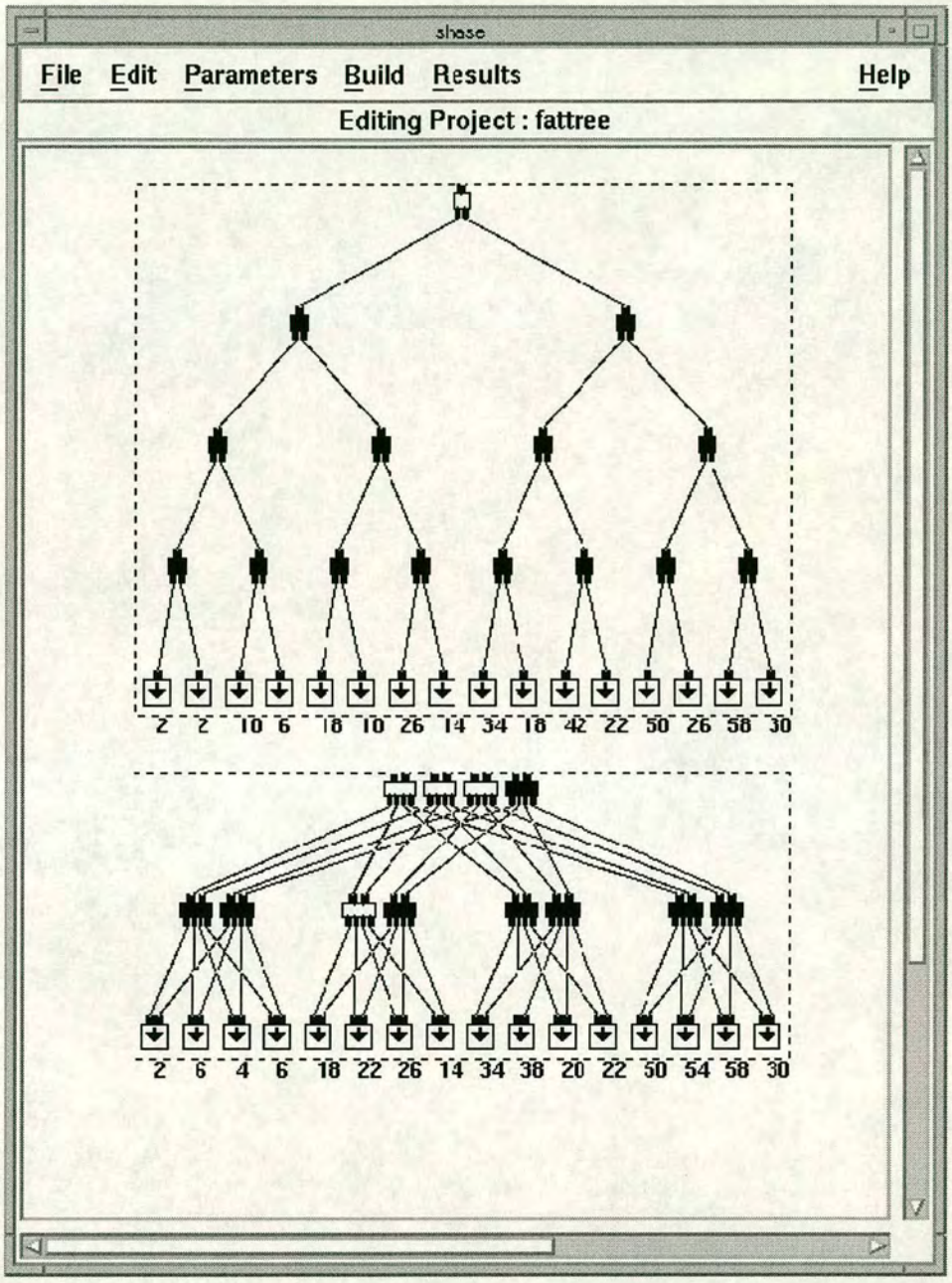


Figure 6.5: A graphical representation of two architectures; a binary tree (top) and a fat tree (bottom).

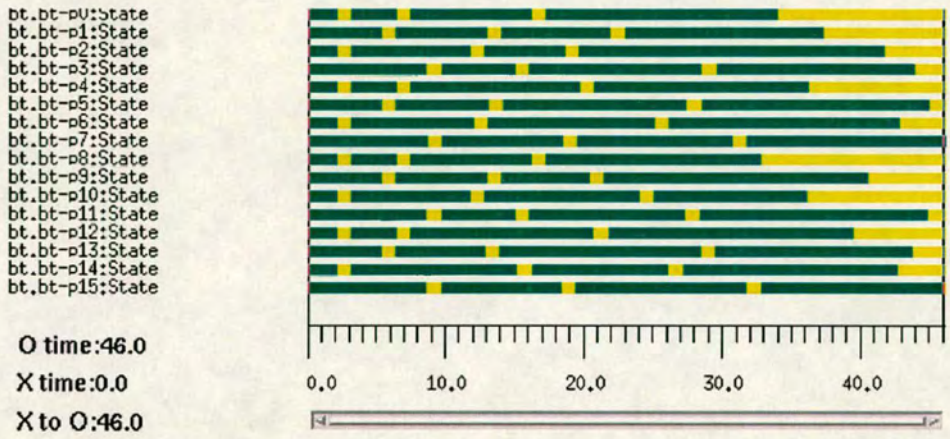


Figure 6.6: A timing diagram showing the detailed behaviour of a binary tree implementing an **allreduce** communications operation.

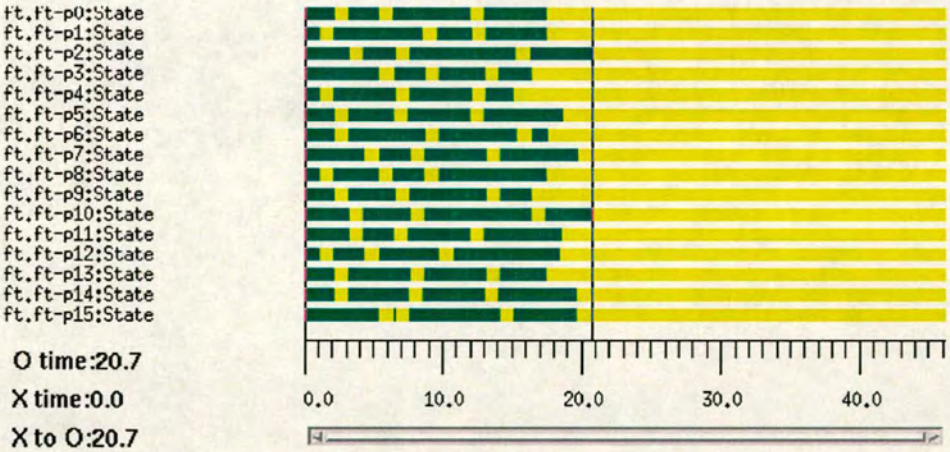


Figure 6.7: A timing diagram showing the detailed behaviour of a fat tree implementing an **allreduce** communications operation.

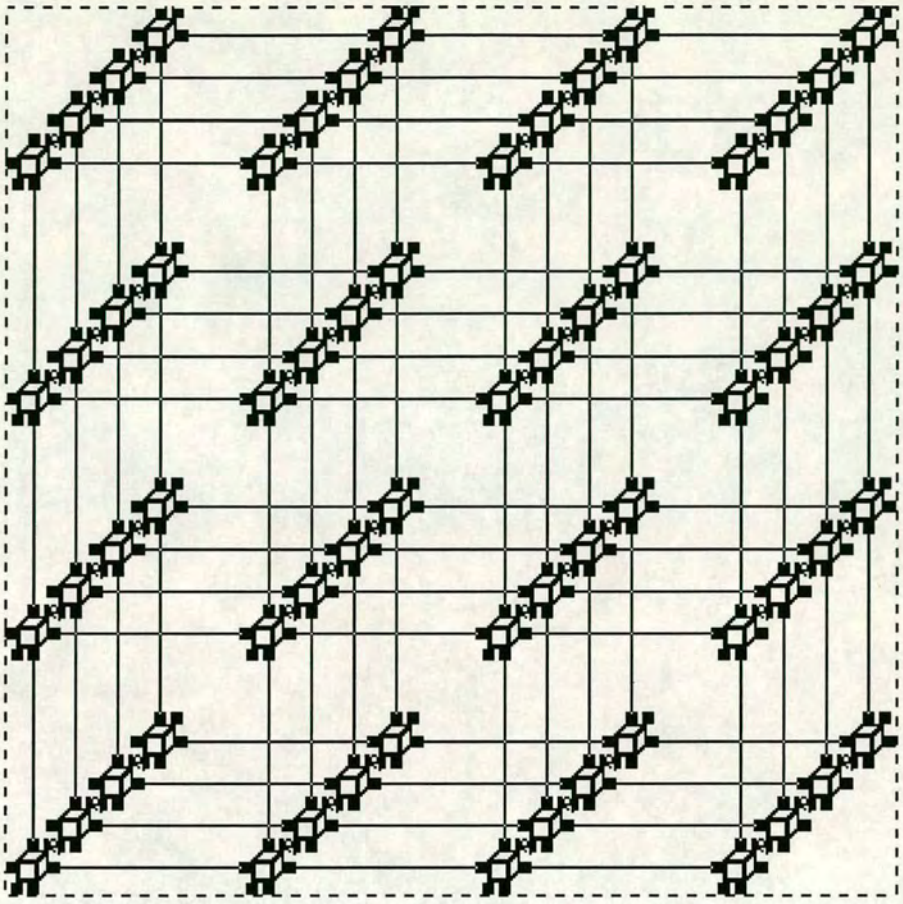


Figure 6.8: A HASE simulation of a 3d torus.

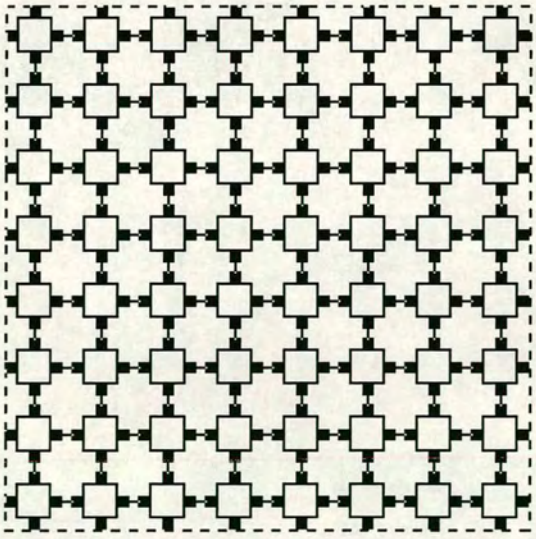


Figure 6.9: A HASE simulation of a 2d torus.

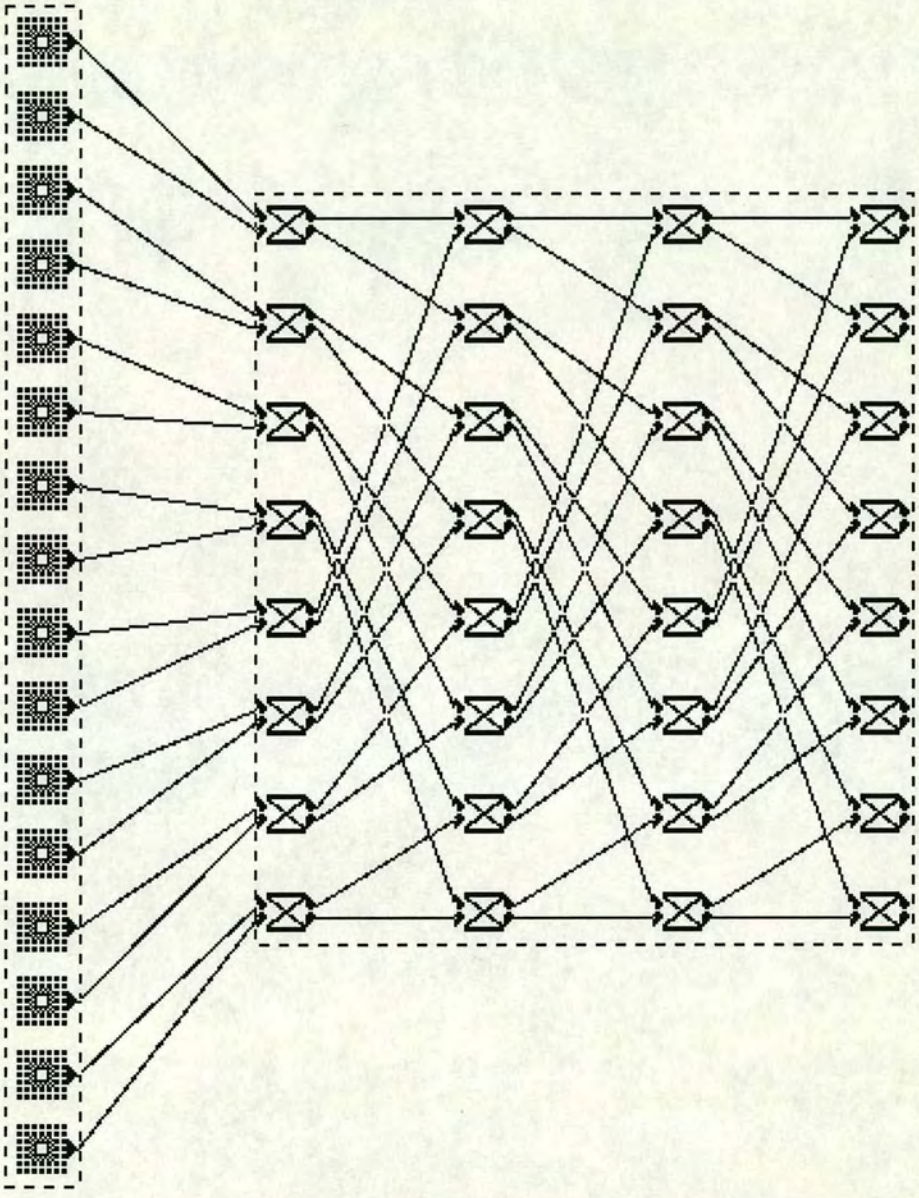


Figure 6.10: A HASE simulation of an omega network.

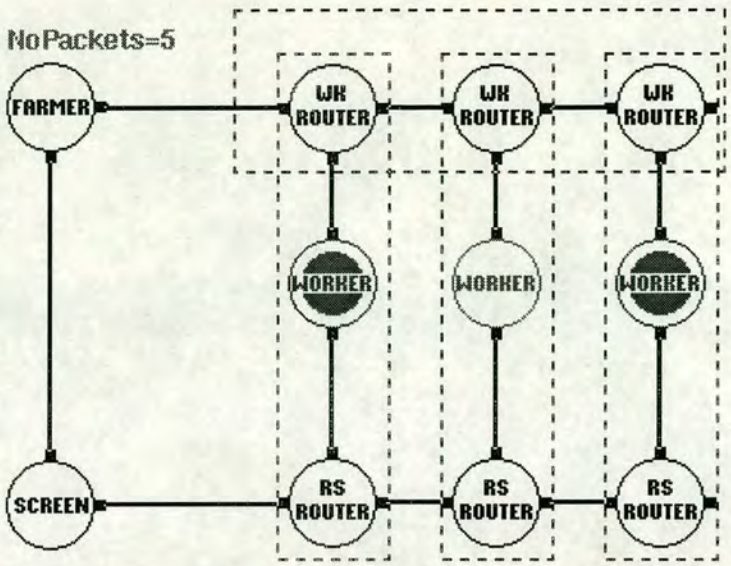


Figure 6.11: A task farm.

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 [4] 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.

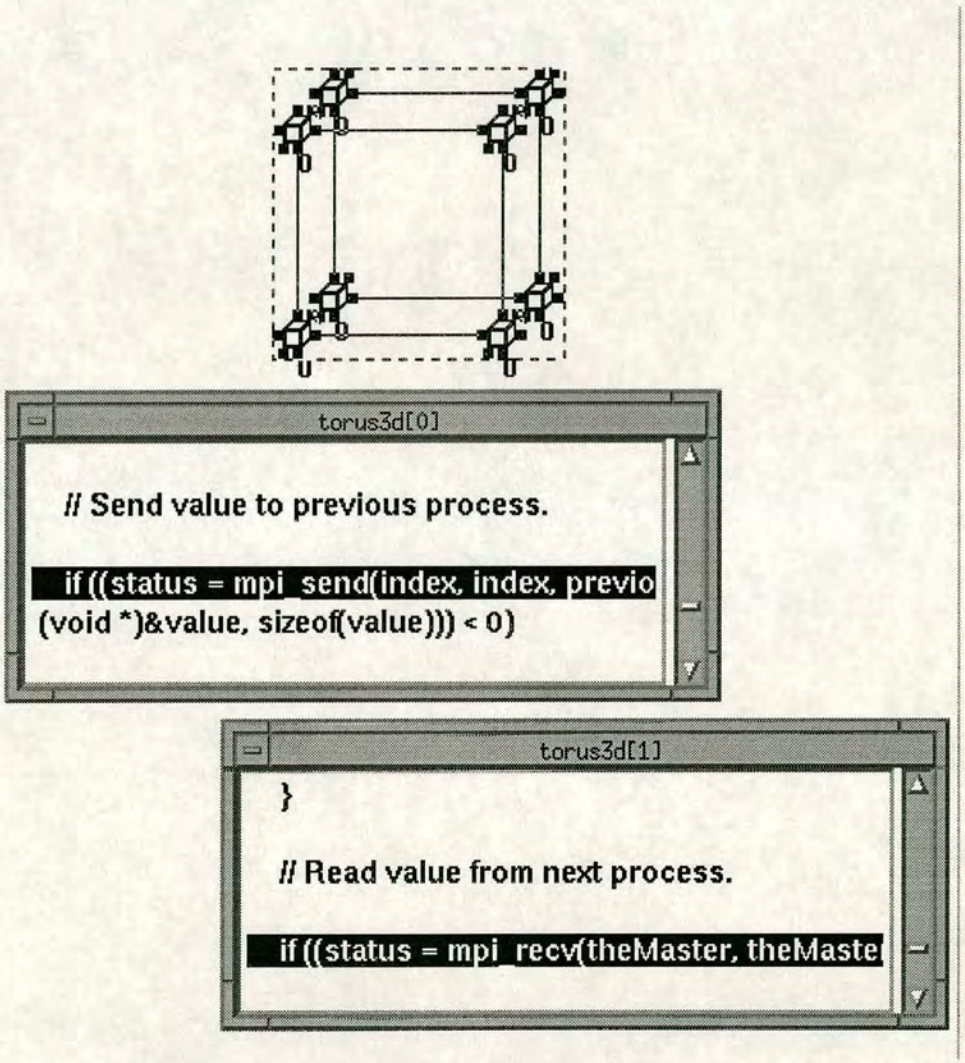


Figure 6.12: Source line highlighting.

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++ [56] 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 [15] 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 [57] 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 [11] 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.

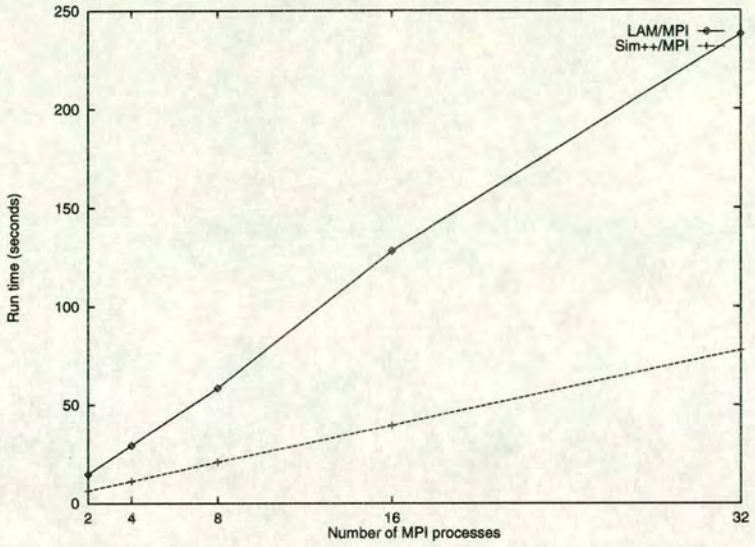


Figure 6.13: Pingpong run times on LAM/MPI and Sim++/MPI

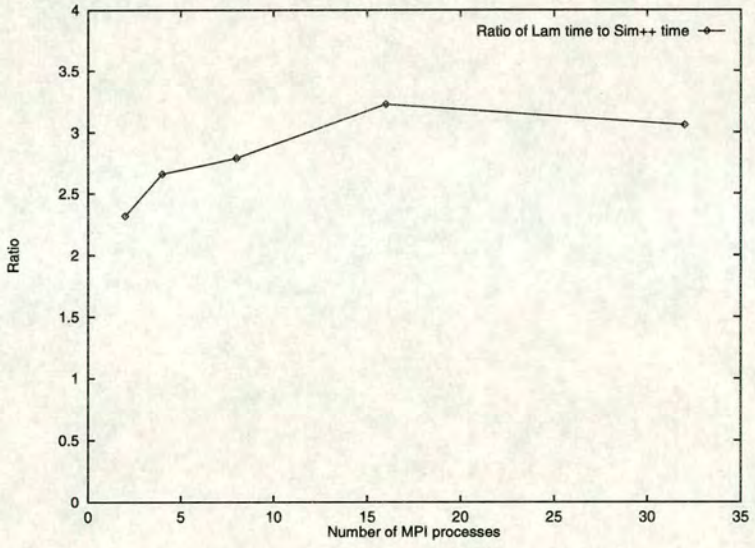


Figure 6.14: Ratio of LAM/MPI to Sim++/MPI run time

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.

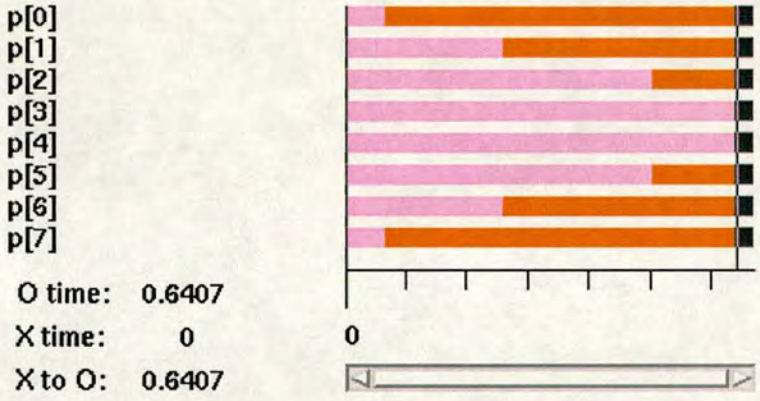


Figure 6.15: Predicted timing diagram for the `mandel` routine for 8 processors on the Cray T3D

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.

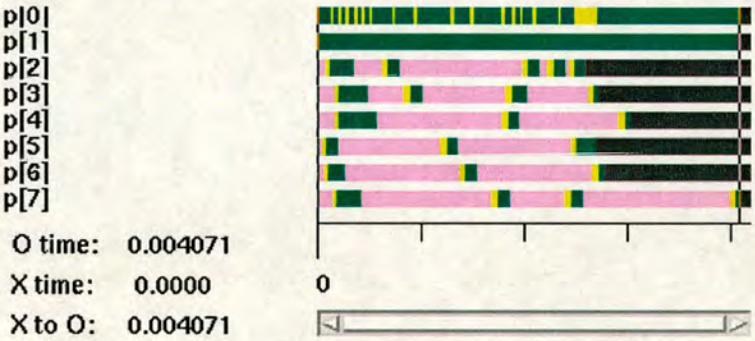


Figure 6.16: Predicted timing diagram of non-deterministic application for Cray T3D.

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

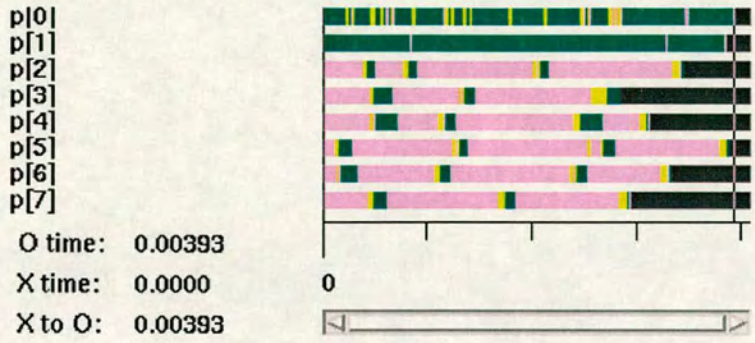


Figure 6.17: Measured timing diagram of non-deterministic application on Cray T3D.

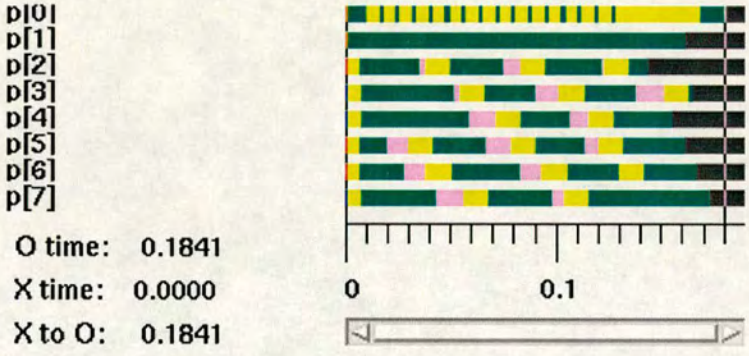


Figure 6.18: Predicted timing diagram of non-deterministic application for network of workstations.

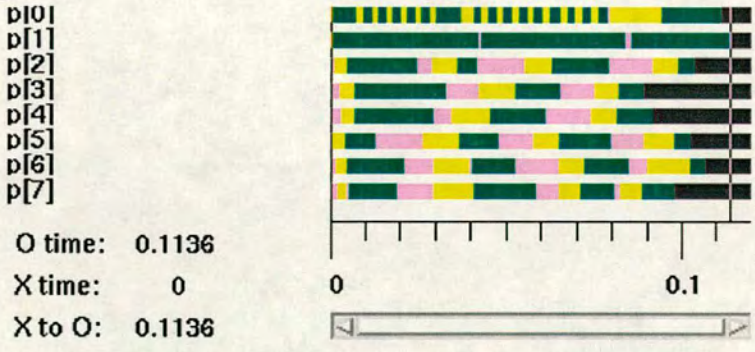


Figure 6.19: Measured timing diagram of non-deterministic application on network of workstations.

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.

Chapter 7

Conclusion

Several methods for attacking the central problem of designing explicitly parallel programs have been presented.

The techniques have focussed on solving the *low level* aspects of parallel program design rather than in creating higher level abstractions. This is because the low level problems have not been solved adequately and higher level programming models are all built on the low primitives. The message passing model of MPI was used.

The main difficulty is developing a technique which is simple enough to use at the initial stages of design yet is accurate enough to provide meaningful guidance. The main competition for any tool for parallel program design is not so much an alternative tool, rather the current situation where performance is left as a "tuning" task to be done after the event. Is it so bad that this aspect is left to tuning? Is design important? In some ways the answer is no. Since software is (superficially) easy to change, why not just build a program one way, test it then make design changes afterwards? In other ways, the fact that the performance characteristics of the primitives are not given means that the program designer is forced to make decisions which affect the performance with nothing other than guesswork and intuition for guidance. It is like designing a circuit with no data sheets.

So the initial phase of work was to provide "data sheets" for programmers (chapter 3). These may be used to provide concrete data to help with pencil and paper calculations at the initial stages of design. Alone, these may be sufficient for many people. A characterisation program generates the sheets automatically for an MPI implementation. It times all the MPI functions using a range of data and machine sizes, then fits a curve to the data. The aim of the data sheets is to describe the delays as seen by the programmer and not to characterise the hardware performance. Thus the time for an `MPI_Send` is quoted as the time

a process is delayed by calling `MPI_Send` (and not the time for the message to arrive). Sheets have been generated for a network of workstations, the Cray T3D and the IBM SP2.

How can the information in the data sheets be best used? This was addressed in chapter 4 which used the raw data from such simple models along with a graphing package to produce scalability plots from equations. This is a very quick method for obtaining rough estimates. The method produces useful graphs showing how much (if any) speedup is expected. The shapes of the expected speedup curves are very similar to those measured on the Cray T3D and a network of workstations. It is easy to see the effects of varying input parameters on a program's overall performance; for example computation time is only predictable to an order of magnitude so speedup curves at both ends of the compute time range can be produced. The restrictions of the technique are that the models are generated by hand, and it is hard to incorporate data dependent communications.

For more complicated patterns of communication, or where more detail is needed, the reverse profiling technique of chapter 5 provides performance prediction using the MPI profiling interface. This applies the data sheet model to programs in the development stage to produce timing diagrams for a single run or scalability graphs for multiple runs. The attraction of this technique is its ease of use. Predictions may be obtained as part of normal development. It is most appropriate for producing timing diagrams showing the detailed behaviour of a single run. Cacheing effects mean that compute time may only be estimated to a factor of ten, but communications time is predicted to a factor of two. The program's exact data dependent communications patterns are incorporated into the expected timing diagram, as long as there are no non-deterministic receives.

Non-deterministic programs may be handled using discrete event simulation. Chapter 6 described a version of this approach. It is a direct execution simulator which uses the running application to drive the simulator kernel. It generates predicted timing diagrams, and because it maintains strict ordering of simulation events it is able to handle non-determinism correctly. The simulator implements low level message passing two to three times faster than implementations of MPI on a single workstation. Because it is a sequential simulator, however, the time to simulate a program running on a parallel machine grows with the number of processors simulated. The MPI data sheets provide the communications model used by the simulator. In addition to these models, simulation allows more detailed models of network architectures to be specified, and some experiments were conducted using graphical techniques to keep the models visible. The cycle count-

ing technique was also used to obtain more accurate estimates of compute times. However it was found to be too cumbersome for widespread use. The simulation approach provides the most detailed results and similar techniques have been suggested by others for parallel program development. However it is too detailed for most developers and it requires a re-implementation of the message passing interface rather than simply building on top of an existing one.

7.1 Prediction as part of design?

In the introduction, it was stated that the ideal was to move away from post-mortem techniques for performance analysis towards incorporating performance into the design stage. From a design point of view, it is better to obtain evaluations of proposed solutions at an early stage of development rather then when coding is completed. The lightweight pencil and paper and graphing techniques may be applied without having to realise the design as a concrete implementation, so fit naturally into the early stages of design to help choose between alternative strategies. The more sophisticated techniques of reverse profiling and simulation both rely on complete programs, or sections of programs, in order to generate more accurate predictions. Thus they are appropriate later in the design cycle for selecting between different key algorithms or determining whether how a program will run on a possibly unavailable machine.

The increase in level of detail of the approaches ties in naturally with top down design, since an appropriate prediction technique may be used at each stage of refinement. At the simplest level, overall estimated timings for application phases may be used. The few phases expected to take the majority of the time may be analysed using a more detailed method. For all the MPI programs developed, the application phases were separated with some form of global communication or synchronisation, so the total time could be calculated by summing the component phase times. This separation of phases (into input, compute and output stages for example) was done in order to obtain correct behaviour of the programs, but also made modular prediction of performance simpler. The BSP model uses the same approach throughout to simplify predictions.

7.2 Further work

7.2.1 Data sheets

Parallel programmers are not given sufficient information at design time to design effective parallel programs. The MPI data sheets presented in chapter 3 go some way towards rectifying this situation for message passing, but similar measurements should be available for other programming models.

The design of the MPI data sheets themselves could be improved, possibly expanding the summary section at the start to include sample times for "common" data and machine sizes in order to save having to plug values into an equation. The current data sheet generator could be expanded to characterise I/O times in addition to the communications functions. It could also characterise a range of computation operations to improve estimates of computation times.

Such data sheets should be a standard part of parallel library documentation.

7.2.2 Combining reverse profiling and simulation

The reverse profiler could be extended by including a parallel simulation engine such as that used in Lapse [11]. This would combine the ease of use of reverse profiling with the ability to handle non-deterministic routines.

7.2.3 Improving compute time prediction

The compiler, processor pipelining and memory hierarchy all conspire to make compute time unpredictable at design time. The only foolproof methods are measurement and full simulation but neither is convenient to do at design time. Intermediate techniques based on cycle counting of assembler code or interpretation of compiler parse trees are too tied to particular implementations to be generally applicable, and in any case are prone to order of magnitude errors.

So it is only practical to predict compute times to within an order of magnitude. The techniques of this thesis left the basic compute time step as a parameter to allow early experimentation to check how sensitive an algorithm is to such compute time variations. In practice, many of the algorithms run on the Cray and the network of workstations produced remarkably little change in expected speedup. They were either communications dominated to an extent that only minimal speedups were available, or computation dominated, giving reasonable speedups across the compute time range. It was only for algorithms with roughly equal computation and communications times that getting the computation step right was essential.

7.3 Overall conclusion

This thesis has presented three approaches to performance prediction; each has its merits. The best technique to use is the simplest one possible. The information in the data sheets along with a calculator (or pen) may well be enough for simple programs. The graphing package is not much more difficult to use for estimates of speedups. For producing timing diagrams showing the way in which complex data dependent communications will work in practice, reverse profiling is as simple to use as standard profiling. Simulation is overkill at the early stages of design, but is appropriate for non-deterministic applications, or for investigating the effects of a program on a network.

Bibliography

[1] A.Dunlop, E.Hernandez, O.Naim, T.Hey, and D.Nicole. A Toolkit for Optimising Parallel Performance. In *HPCN International Conference Milan*, number 919 in LNCS, pages 548–553. Springer-Verlag, May 1995.

[2] A. Agarwal. Limits on interconnection network performance. *IEEE Trans. on Par. & Dist. Sys.*, 2(4), October 1991.

[3] 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.

[4] 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.

[5] 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.

[6] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogenous systems. *Int. Journal of Comp. Sim.*, August 1992.

[7] G.D. Burns, R.B. Daoud, and J.R. Vaigl. LAM: An Open Cluster Environment for MPI. In *Supercomputing Symposium '94*, Toronto, Canada, June 1994.

[8] L.J. Clarke. PUL concepts I. Technical Report EPCC-KTP-PUL-CONC-I, Edinburgh Parallel Computing Centre, University of Edinburgh, 1991.

[9] M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman & MIT Press, 1989.

[10] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* San Diego, CA, May 1993.

[11] 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.

[12] M.A. Driscoll and W.R. Daasch. Accurate predictions of parallel program execution time. *Journal of Parallel and Distributed Computing*, 25(1), February 1995.

[13] T. Fahringer and H.P. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.

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

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

[16] I. Foster. *Designing and Building Parallel Programs*, chapter 3. Addison-Wesley, 1994. Available online at http://www.mcs.anl.gov/dbpp/.

[17] F.W. Howell. MPI Data Sheet Generation Routines. `http://www.dcs.ed.ac.uk/home/fwh/perfchar`, 1996.

[18] J.L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[19] F. Hartleb. Graph models for Performance Evaluation of Parallel Programs. In A.Bode and M.Dal Cin, editor, *Parallel Computer Architectures : Theory, Hardware, Software, Applications*, number 732 in LNCS. Springer-Verlag, 1993.

[20] M.T. Heath and J.A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, Sept 1991.

[21] T. Hey, J. Dongarra, and R. Hockney. PARKBENCH: Parallel kernels and benchmarks. available at `http://www.netlib.org/parkbench/html/`, 1996.

[22] J.M.D. Hill, P.I. Crumpton, and D.A. Burgess. Theory, practice and a tool for bsp performance prediction. Technical Report TR-4-96, Oxford University Programming Research Group, Feb 1996.

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

[24] J. Hillston. PEPA: Performance Enhanced Process Algebra. Technical Report CSR-24-93, Department of Computer Science, University of Edinburgh, March 1993.

[25] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[26] A. Hondroudakis and R. Procter. The design of a tool for parallel program performance analysis and tuning. In *Proceedings of IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems.* IFIP, April 1994.

[27] F.W. Howell. Reverse profiling. 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 244–255. IFIP, Chapman and Hall, March 1996.

[28] 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.

[29] 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.

[30] H.Wabnig, G.Haring, D.Kranzmuller, and J.Volkert. Communication Pattern Based Performance Prediction on the nCUBE-2 multiprocessor System. In *CONPAR*, pages 41–52, Linz, Austria, Sept 1994. Springer-Verlag.

[31] 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.

[32] 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.

[33] J.P.Singh and J.L.Hennessy. Finding and exploiting parallelism in an ocean simulation program: experience, results and implications. *Journal of parallel and distributed computing*, 15:27–48, 1992.

[34] K. Ciula. PVM and MPI benchmarks on the LACE cluster. `http://www.lerc.nasa.gov/WWW/ACCL/PARALLEL/benchmark.html`, 1995.

[35] K.B. Kenny and K. Lin. Building flexible real-time systems using the flex language. *IEEE Computer*, 24(5):70–78, May 1991.

[36] P.J.B King. *Computer and Communication Systems Performance Modelling.* Prentice-Hall, 1990.

[37] De-Ron Liang and S.K. Tripathi. Performance prediction of parallel computation. In *Proc 8th IPPS*, pages 625–629. CS Press, 1994.

[38] N. MacDonald. Predicting execution times of sequential scientific kernels. In Christoph W. Kessler, editor, *Automatic Parallelization*, pages 32–44. Vieweg, 1994.

[39] V. Mak and S. Lundstrom. Predicting performance of parallel computations. *IEEE trans. on par. & distr. sys.*, 1(3), July 1990.

[40] M. Marr. Equations for describing MPI run times. (private communication), April 1995.

[41] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[42] F.M. Moller. A temporal calculus of communicating systems. Technical Report ECS-LFCS-89-104, Univ. of Edinburgh Dept. of Computer Science, 1989.

[43] MPICH - A Portable Implementation of MPI. `http://www.mcs.anl.gov/Projects/mpi/mpich/`, 1996.

[44] NASA AMES Research Center. AIMS: An Automated Instrumentation and Monitoring System. `http://www.nas.nasa.gov/NAS/Tools/Projects/AIMS/`, 1995.

[45] S. Nog and D. Kotz. Performance Comparison of TCP/IP and MPI on FDDI, Fast Ethernet and Ethernet. Technical Report PCS-TR95-273, Department of Computer Science, Dartmouth College, Hanover, NH, November 1995. Available at `http://www.cs.dartmouth.edu/reports/abstracts/PCS-TR95-273.html`.

[46] C.Y. Park and A.C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, May 1990.

[47] J.L. Peterson. *Petri-net theory and the modelling of systems.* Prentice Hall, 1981.

[48] P.Pouzet, J.Paris, and V.Jorrand. Parallel Application Design: The Simulation Approach with HASTE. In W.Gentzsch and V.Harms, editors, *High Performance Computing and Networking II : Networking and Tools*, number 797 in LNCS, pages 379–393. Springer-Verlag, April 1994.

[49] P. Puschner and C.H. Koza. Calculating the maximum execution time of real-time programs. *J. Real-Time Systems*, 1(2):159–176, 1989.

[50] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz, and Luis F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In Anthony Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference.* IEEE Computer Society, 1993.

[51] 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.

[52] R.W.Numrich, P.L.Springer, and J.C.Peterson. Measurement of Communication Rates on the Cray T3D Interprocessor Network. In W.Gentzsch and V.Harms, editors, *High Performance Computing and Networking II : Networking and Tools*, number 797 in LNCS, pages 150–157. Springer-Verlag, April 1994.

[53] R.H. Saavedra, R.S. Gaines, and M.J. Carlton. Micro benchmark analysis of the KSR1. In *Supercomputing '93*, Portland, Oregon, 1993.

[54] R.H. Saavedra-Barrera, A.J. Smith, and E. Miya. Machine characterisation based on an abstract high-level language machine. *IEEE Trans. on Comp.*, 38(12):1659–1679, December 1989.

[55] S.R. Sarukkai. Scalability analysis tools for SPMD message-passing parallel programs. In *MASCOTS '94: Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, January 1994.

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

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

[58] Sivan Toledo. *Quantitative Performance Modelling of Scientific Computations and Creating Locality in Numerical Algorithms.* PhD thesis, Massachusetts Institute of Technology, 1995. Also available as Technical Report MIT-LCS-TR-656.

[59] Sivan Toledo. Performance prediction with benchmaps. In *Proceedings of the 10th International Parallel Processing Symposium, Honolulu, Hawaii*, pages 479–484, Los Alamitos, California, April 1996. IEEE, IEEE Computer Society Press.

[60] L.G. Valliant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[61] Gregory V. Wilson. Assessing the Usability of Parallel Programming Systems: The Cowichan Problems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems.* Birkhäuser Verlag AG, April 1994.

Appendix A

An overview of MPI

This appendix gives a brief overview of the MPI functions referred to in the thesis. For a full description see [15].

MPI_Barrier Performs a barrier synchronisation amongst a group of processes.

MPI_Bcast Broadcasts data from the root node to all processes in the group.

MPI_Reduce Reduces a set of data items held on separate processes down to a single value on the root process. The operation for the reduction can be summation, minimum/maximum, or provided as a function by the programmer.

MPI_Allreduce As reduce, but the answer is returned to all processes rather than just the root.

MPI_Scatter Scatters data from the root process to all processes in the group.

MPI_Gather Gathers data from all processes to the root process.

MPI_Allgather Gathers data from all processes to all processes.

MPI_Alltoall Each process sends and receives distinct data to/from every other process in the group.

MPI_Send Sends a message from one process to another.

MPI_Recv Receives a message from another process.

MPI_Wtime Returns the current local timer value, in seconds.

MPI_Wtick Returns the resolution of MPI_Wtime.

MPI_Comm_split Partitions a group of processes into a set of smaller groups.

Appendix B

An example data sheet

This appendix gives an MPI data sheet generated on the Cray T3D by the routine described in chapter 3.

Datasheets for MPI on Cray T3D

Rawtiming Routine

Timings made : Fri Jul 5 17:28:40 1996

Run time parameters

Iterations	3
Elements	from 1 to 8192
	with log multiplier 2
Data type	int (8 bytes)
Processes	128
Timer resolution	$0.0066\mu s$

Small messages (32 integers or less)

MPI Function	Time (μs)			Goodness of fit (Q)
send	$30 +$	$0 \times ndata$		0.94
ssend	$80 +$	$0.5 \times ndata$		1
rsend	$30 +$	$0.1 \times ndata$		1
isend1	$30 +$	$0.2 \times ndata$		1
isend2	$10 +$	$0 \times ndata$		0.9
isendoverlap	$3 +$	$0 \times ndata$		0.74
recv	$60 +$	$0.7 \times ndata$		0.97
recvmin	$30 +$	$0.9 \times ndata$		0.77
irecv1	$30 +$	$0 \times ndata$		1
irecv2	$40 +$	$0 \times ndata$		0.065
irecvoverlap	$0.6 +$	$0.02 \times ndata$		1
sendrecv	$90 +$	$1 \times ndata$		0.97
pingpong	$100 +$	$1 \times ndata$		0.99
alltoall	$40 +$	$50 \times nprocs +$	$2 \times ndata$	1
allsend	$40 +$	$0.05 \times nprocs +$	$0.09 \times ndata$	1
gather	$70 +$	$10 \times nprocs +$	$0.7 \times ndata$	1
allgather	$40 +$	$40 \times nprocs +$	$1 \times ndata$	1
reduce	$300 +$	$3 \times nprocs +$	$2 \times ndata$	1
allreduce	$300 +$	$6 \times nprocs +$	$2 \times log(nprocs) \times ndata$	1
bcast	$200 +$	$2 \times nprocs +$	$0.6 \times ndata$	1

Large messages (32 integers or more)

MPI Function	Time (μs)	Goodness of fit (Q)
send	$30 + 0.09 \times ndata$	0.98
ssend	$100 + 0.1 \times ndata$	0.98
rsend	$30 + 0.09 \times ndata$	0.98
isend1	$30 + 0.09 \times ndata$	0.98
isend2	$10 + 0.001 \times ndata$	0.78
isendoverlap	$3 + 0.0002 \times ndata$	0.93
recv	$60 + 0.5 \times ndata$	1
recvmin	$30 + 0.4 \times ndata$	1
irecv1	$40 + 0 \times ndata$	0.44
irecv2	$5 + 0.5 \times ndata$	0.87
irecvoverlap	$0.5 + 0.005 \times ndata$	$3e-05$
sendrecv	$90 + 0.6 \times ndata$	1
pingpong	$100 + 1 \times ndata$	1
alltoall	$0 + 40 \times nprocs + 0.3 \times nprocs \times ndata$	1
allsend	$40 + 0.1 \times nprocs + 0.1 \times ndata$	1
gather	$0 + 200 \times log(nprocs) + 0.0009 \times nprocs^2 \times ndata$	2
allgather	$0 + 40 \times nprocs + 0.3 \times nprocs \times ndata$	1
reduce	$300 + 2 \times nprocs + 0.6 \times log(nprocs) \times ndata$	1
allreduce	$300 + 6 \times nprocs + 1 \times log(nprocs) \times ndata$	1
bcast	$100 + 2 \times nprocs + 0.2 \times log(nprocs) \times ndata$	1

Barrier type routines

MPI Function	Time (μs)	Goodness of fit (Q)
barrier	$10 + 8 \times log(nprocs)$	1

Notes on using this datasheet

This section explains how to use the information in the datasheet. In general the times are given in terms of the delays seen by a process between calling an MPI function and it returning.

Measurements

The parameters used for the timing run are given in the first table. Each measurement was repeated `iterations` times. The mean and significance were estimated from these measurements, and both mean and significance were used by the curve fitting routines. This explains some oddities in the curve fits, since some points are regarded as more significant than others. Message and process sizes are given. The resolution of the timer reported by `MPI_Wtick()` is shown. Minimum and maximum values on the curve fit parameters are included in the detailed data sheets.

send, ssend, rsend

The time the sender takes to execute an `MPI_Send()`, `MPI_Ssend()`, `MPI_Rsend()`. given that the matching receive was started at the same time as the send.

isend1, isend2, isendoverlap

Asynchronous sends are measured with three times; `isend1` is the time to post the send; `isend2` is the time spent waiting for the send to complete; `isendoverlap` is the amount of time available for hiding computation between between posting the send and the send completing. (without extending the total time for the send beyond `isend1+isend2`).

recv

The time the receiver takes to execute an `MPI_Recv()`. (given that the matching send was started at the same time).

recvmin

The minimum time the receiver takes to execute an `MPI_Recv()`. (if the send was started at least 2 recv times before the receive starts).

irecv1, irecv2, irecvoverlap

Asynchronous recvs are measured with three times; `irecv1` is the time to post the recv; `irecv2` is the time spent waiting for the recv to complete (given that a matching send was started concurrently with the irecv); `irevoverlap` is the amount of time available for hiding computation between between posting the recv and the recv completing. (without extending the total time for the recv beyond `irecv1+irecv2`).

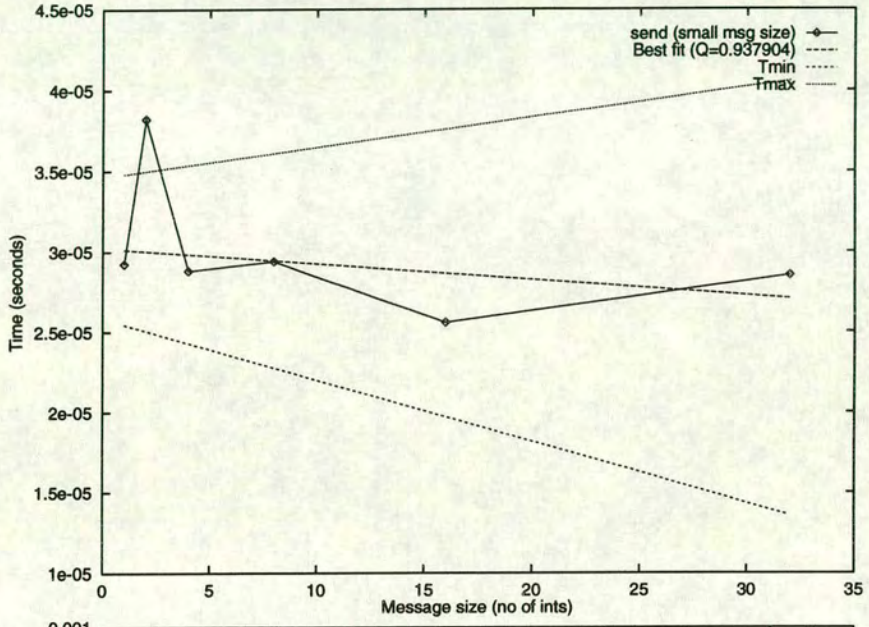
pingpong

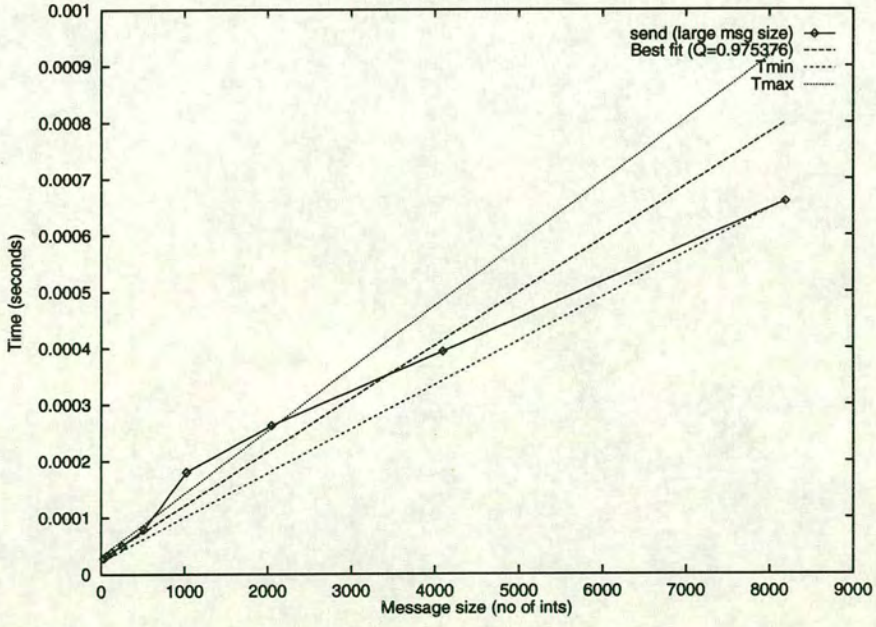
The round trip time to bounce a message between two processes (using standard sends and recvs).

Collective communications

The times for collective communications are measured as the maximum time to execute the function across all processes, assuming the processes are synchronised beforehand. The message size parameter for collective calls is the same as that given in the MPI function call.

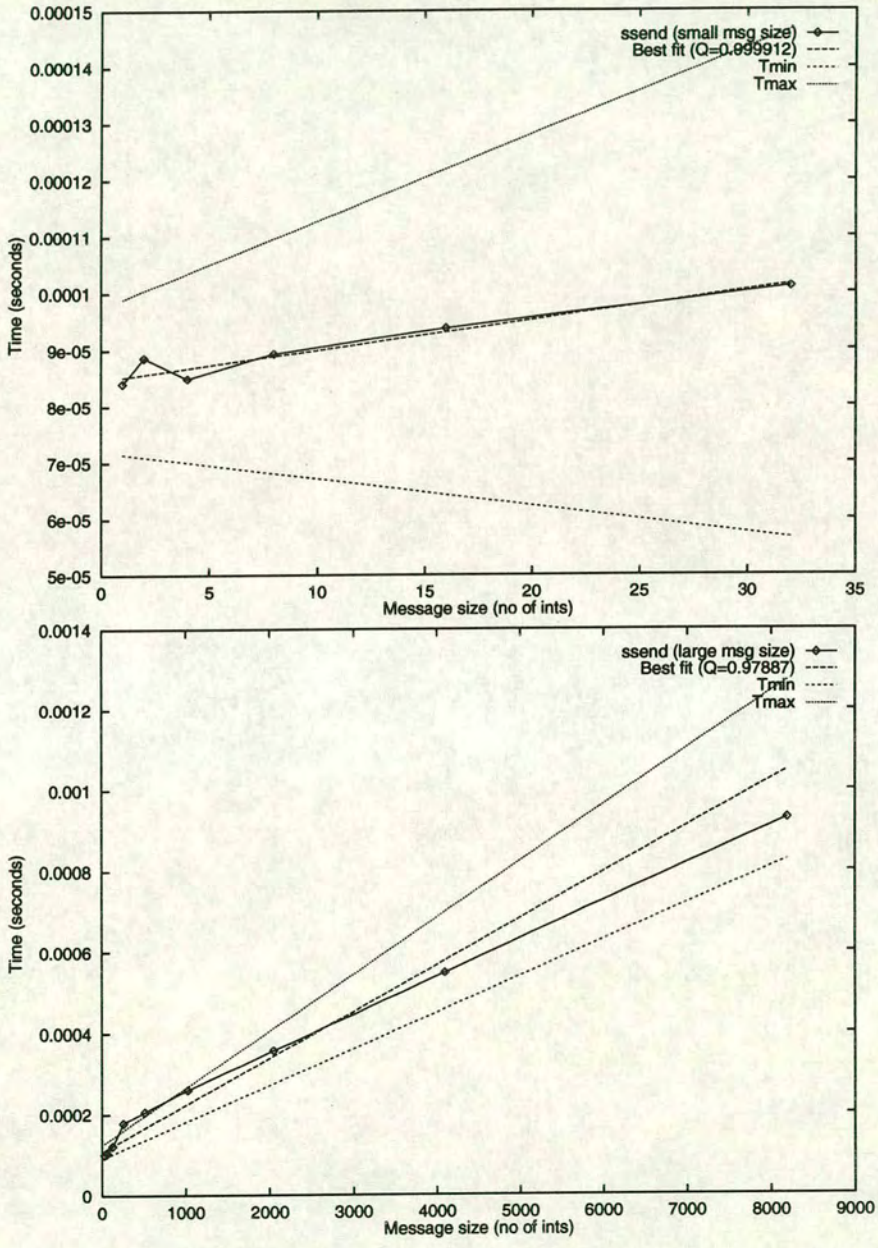
send





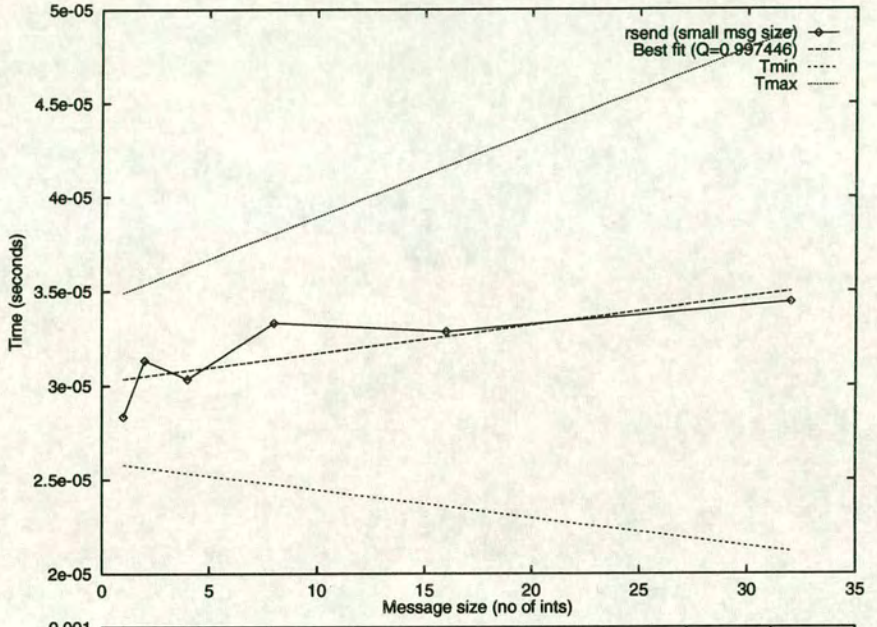
$$T_{send}(\mu s) = \begin{cases} (30 \pm 4) + (0 \pm 0.3) \times ndata & \text{if } ndata <= 32 \\ (30 \pm 5) + (0.09 \pm 0.02) \times ndata & \text{if } ndata > 32 \end{cases}$$

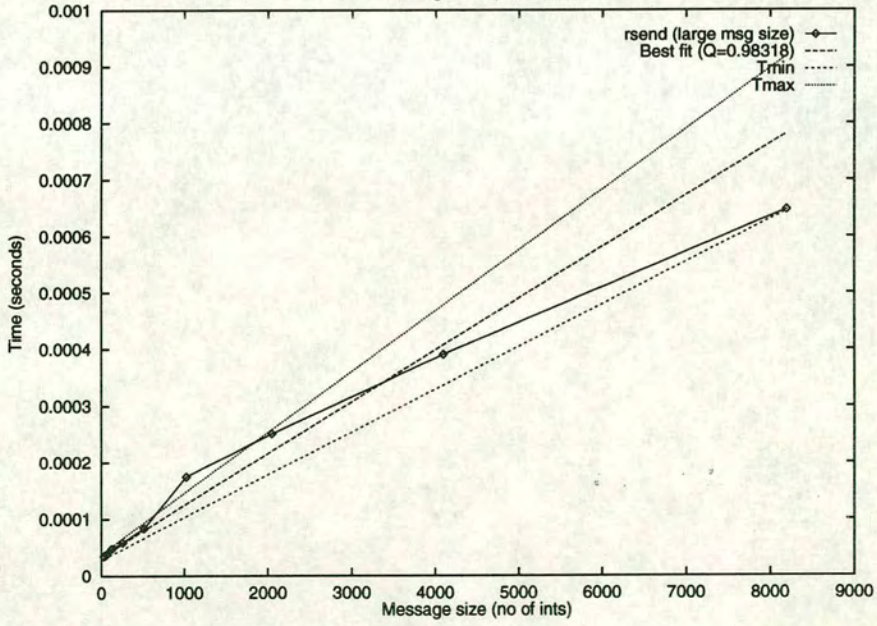
ssend



$$T_{ssend}(\mu s) = \begin{cases} (80 \pm 10) + (0.5 \pm 1) \times ndata & \text{if } ndata <= 32 \\ (100 \pm 20) + (0.1 \pm 0.03) \times ndata & \text{if } ndata > 32 \end{cases}$$

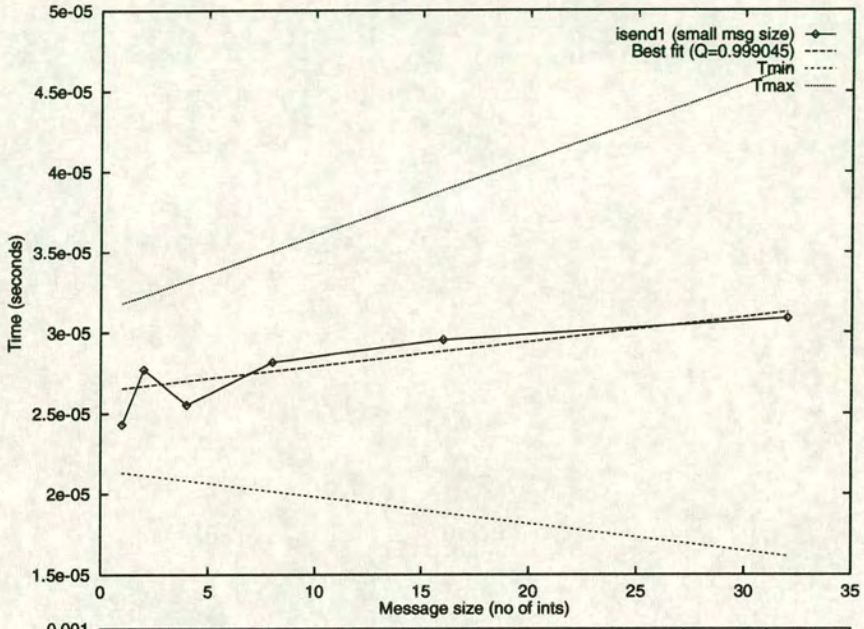
rsend

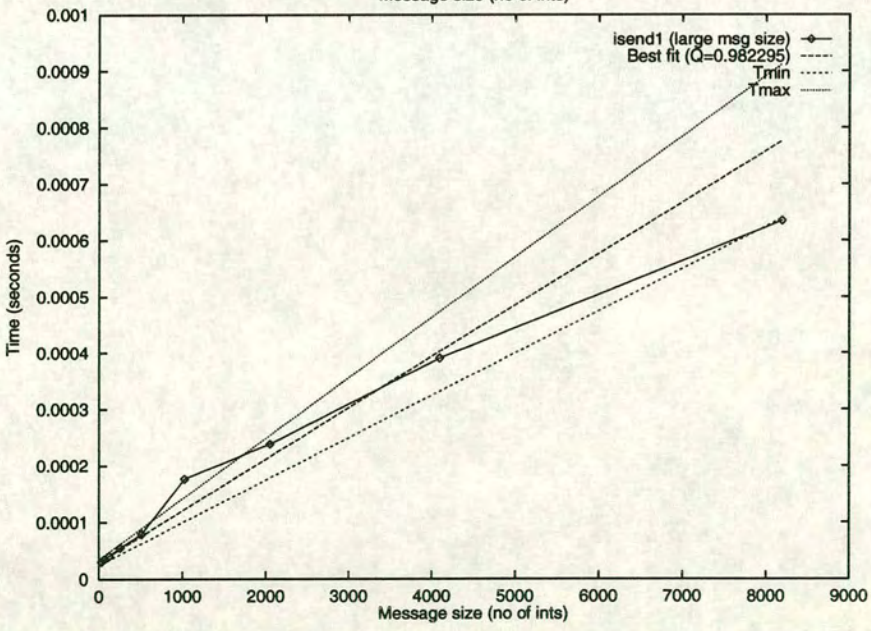




$$T_{rsend}(\mu s) = \begin{cases} (30 \pm 4) + (0.1 \pm 0.3) \times ndata & \text{if } ndata <= 32 \\ (30 \pm 5) + (0.09 \pm 0.02) \times ndata & \text{if } ndata > 32 \end{cases}$$

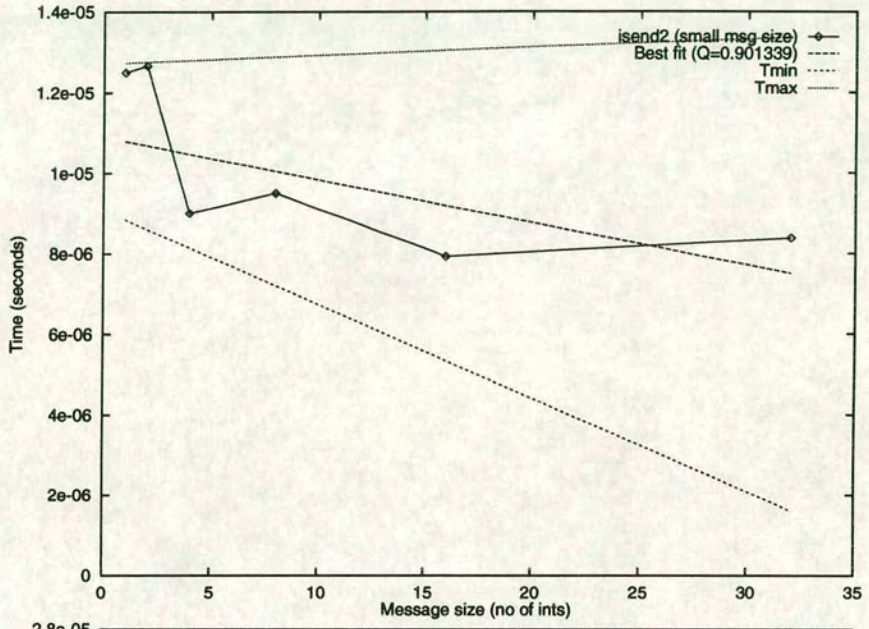
isend1

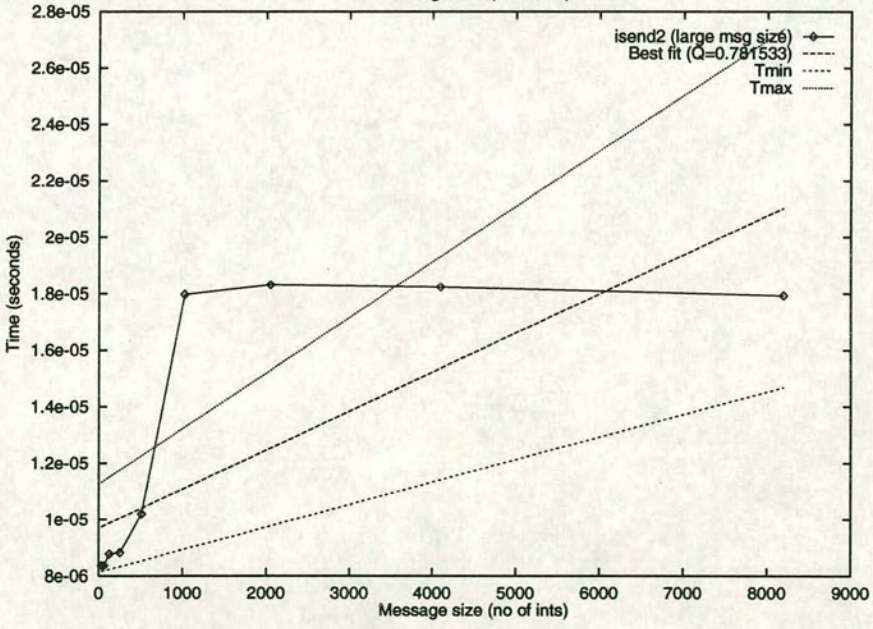




$$T_{isend1}(\mu s) = \begin{cases} (30 \pm 5) + (0.2 \pm 0.3) \times ndata & \text{if } ndata <= 32 \\ (30 \pm 5) + (0.09 \pm 0.02) \times ndata & \text{if } ndata > 32 \end{cases}$$

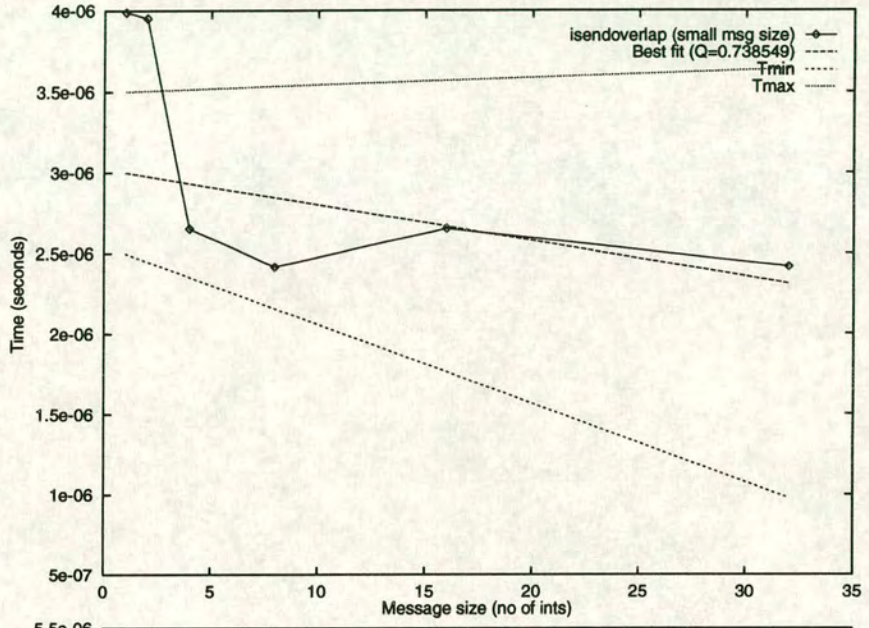
isend2

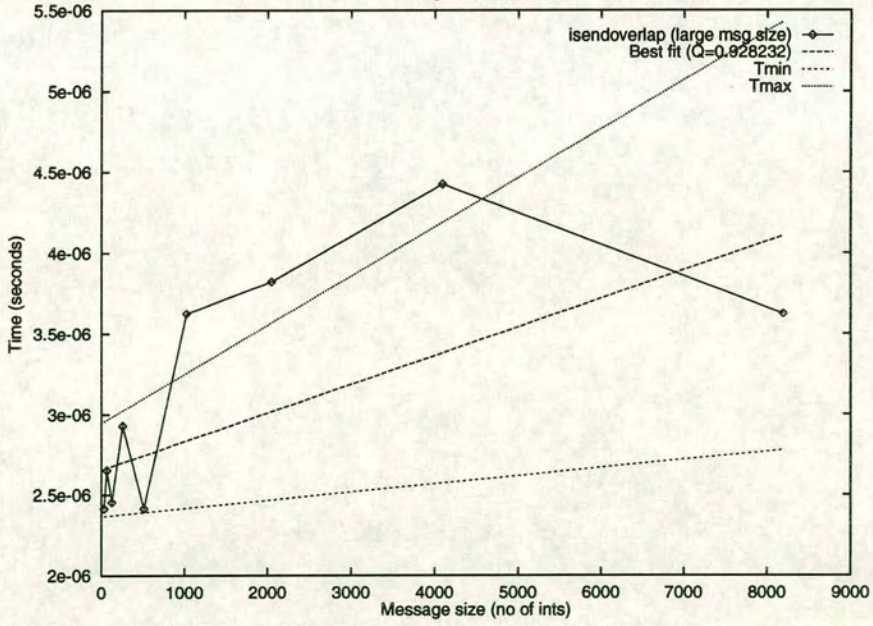




$$T_{isend2}(\mu s) = \begin{cases} (10 \pm 2) + (0 \pm 0.1) \times ndata & \text{if } ndata <= 32 \\ (10 \pm 2) + (0.001 \pm 0.0006) \times ndata & \text{if } ndata > 32 \end{cases}$$

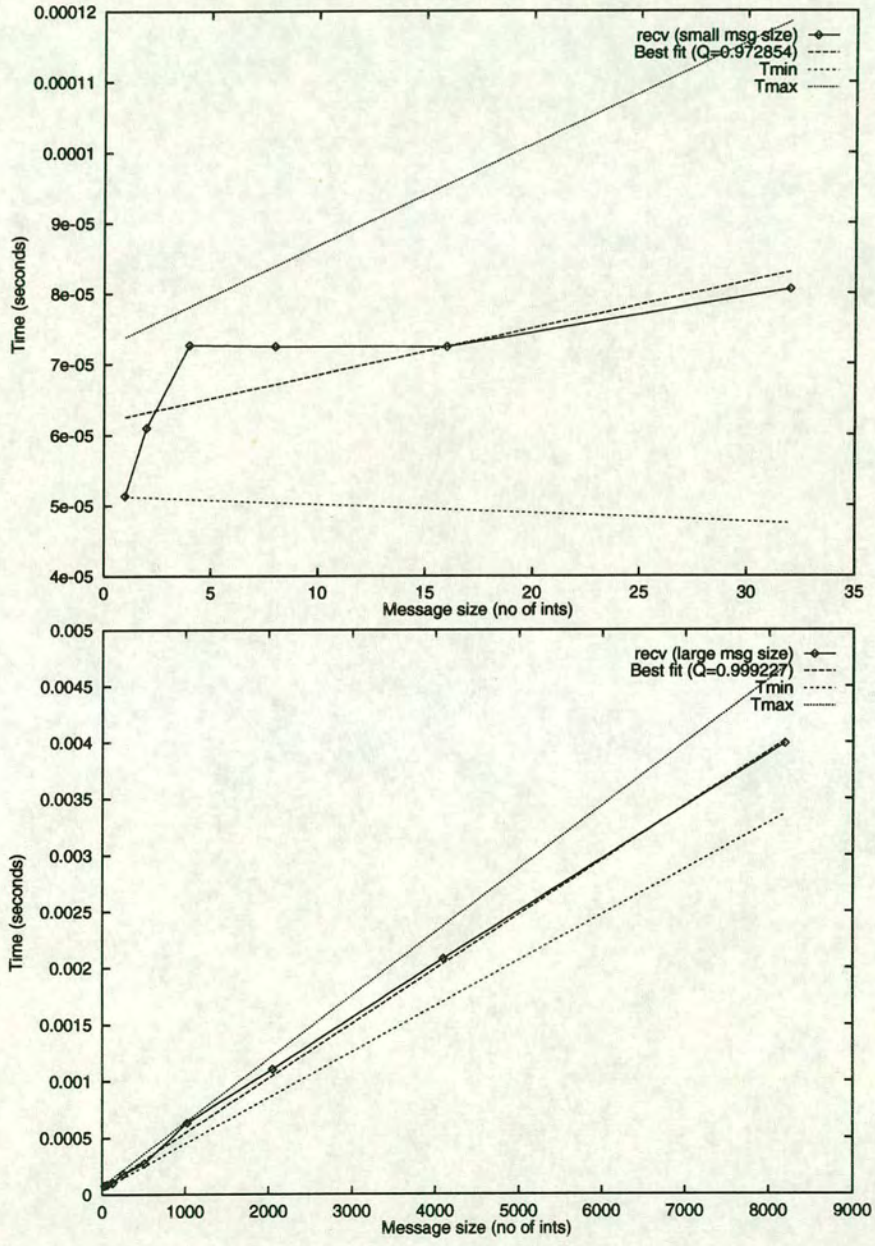
isendoverlap





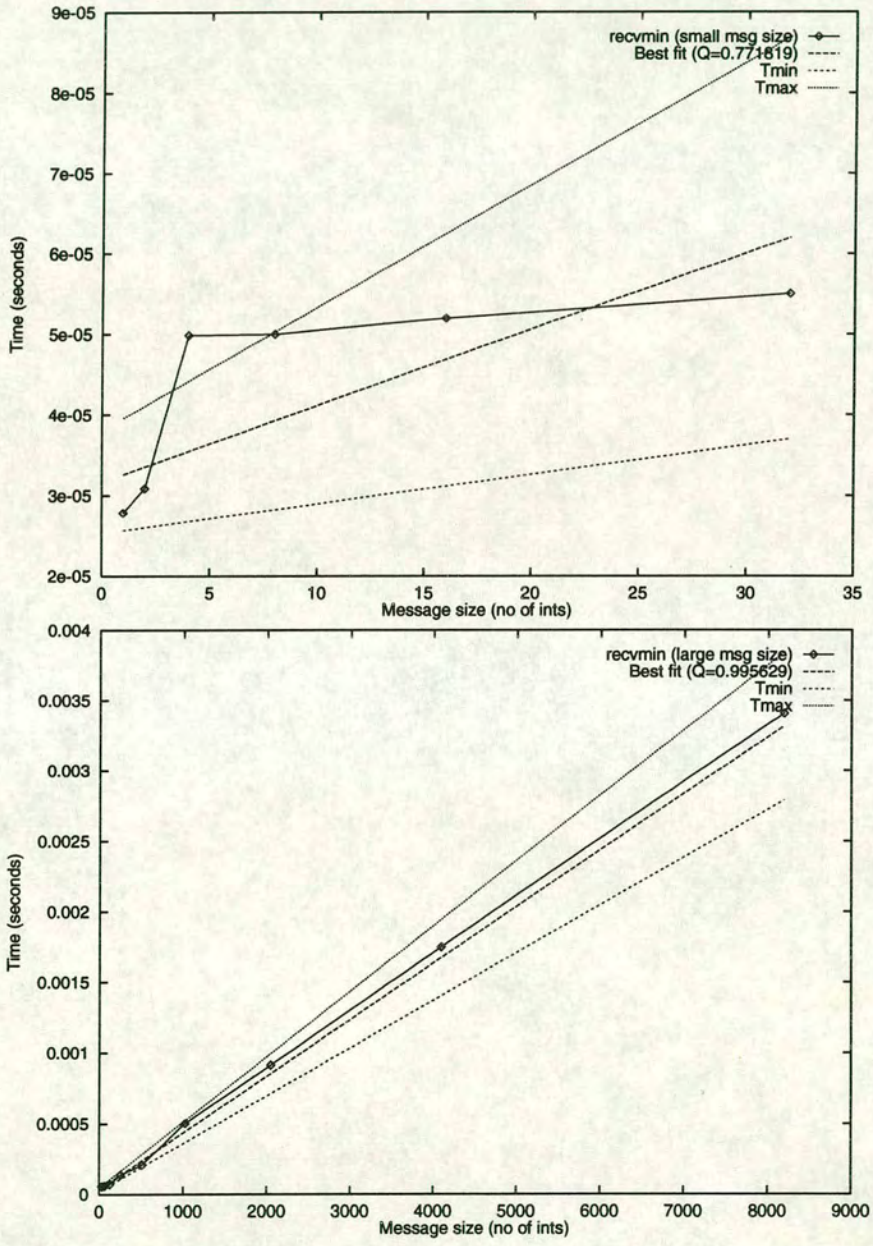
$$T_{isendoverlap}(\mu s) = \begin{cases} (3 \pm 0.5) + (0 \pm 0.03) \times ndata & \text{if } ndata <= 32 \\ (3 \pm 0.3) + (0.0002 \pm 0.0001) \times ndata & \text{if } ndata > 32 \end{cases}$$

recv



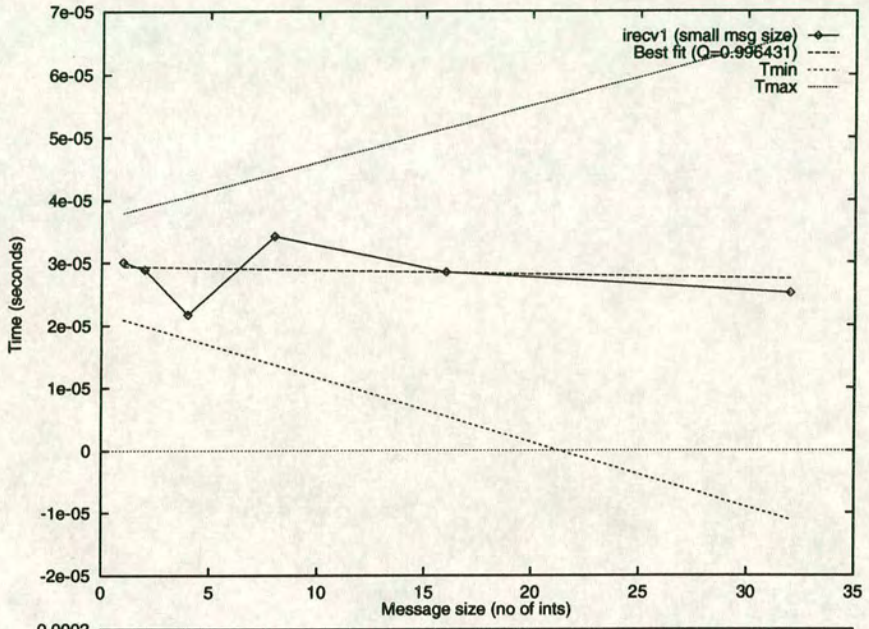
$$T_{recv}(\mu s) = \begin{cases} (60 \pm 10) + (0.7 \pm 0.8) \times ndata & \text{if } ndata <= 32 \\ (60 \pm 20) + (0.5 \pm 0.08) \times ndata & \text{if } ndata > 32 \end{cases}$$

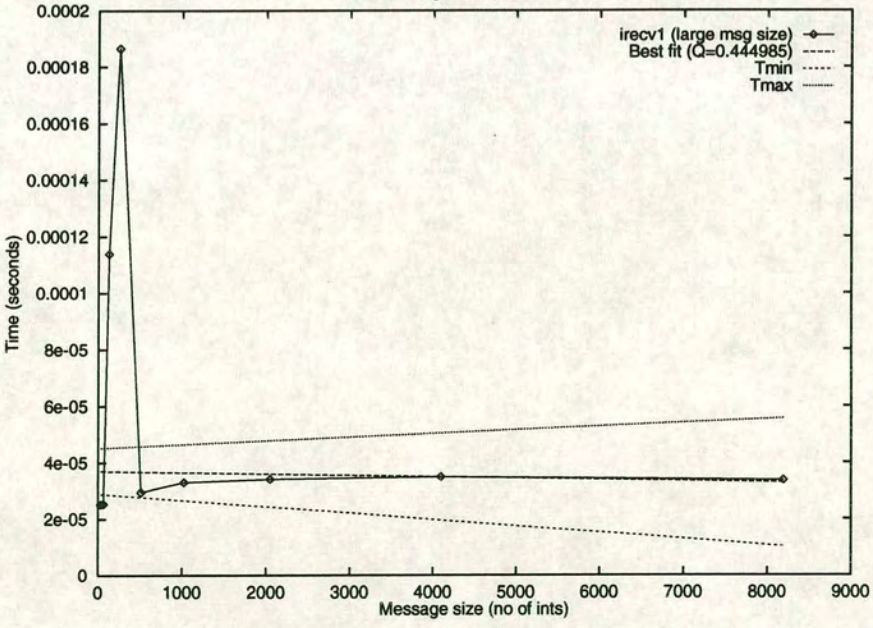
recvmin



$$T_{recvmin}(\mu s) = \begin{cases} (30 \pm 6) + (0.9 \pm 0.6) \times ndata & \text{if } ndata <= 32 \\ (30 \pm 10) + (0.4 \pm 0.06) \times ndata & \text{if } ndata > 32 \end{cases}$$

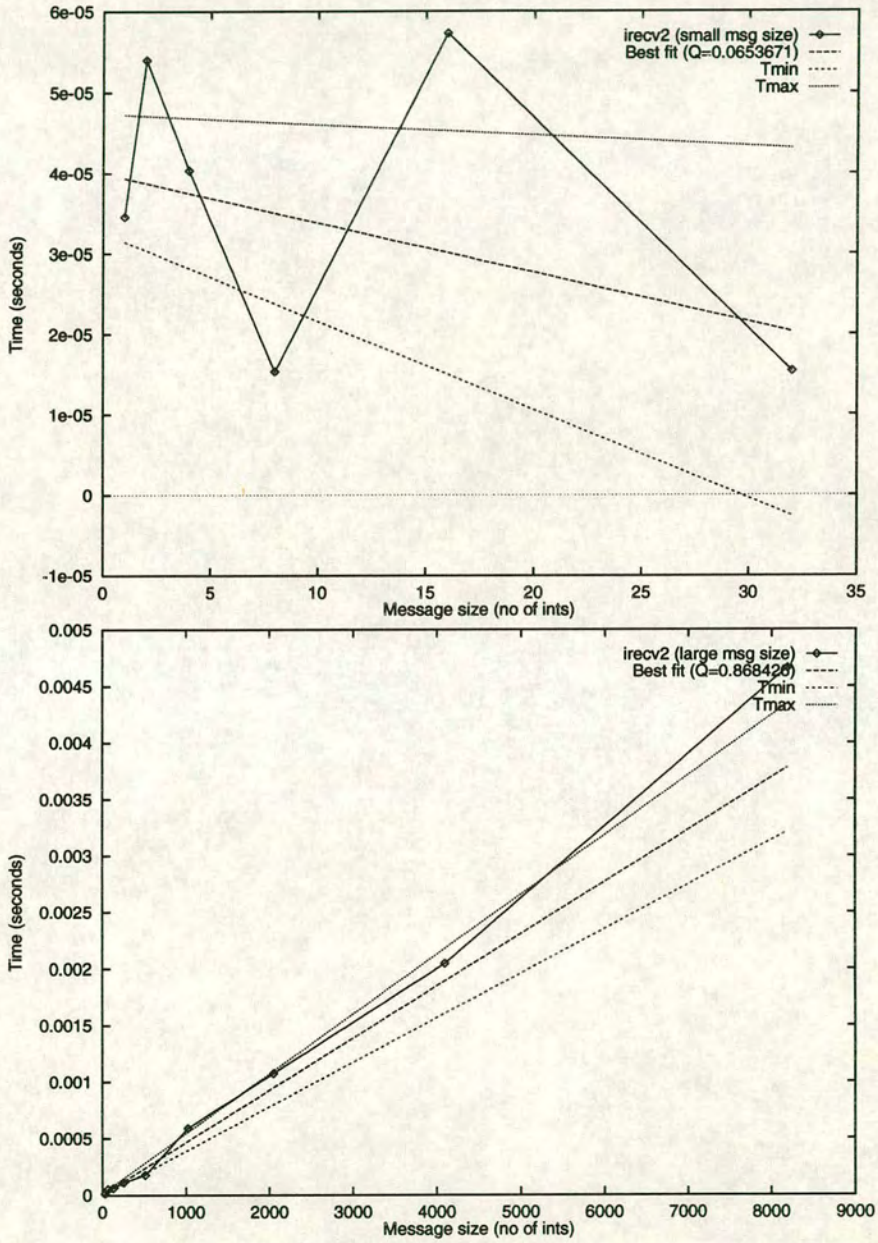
irecv1





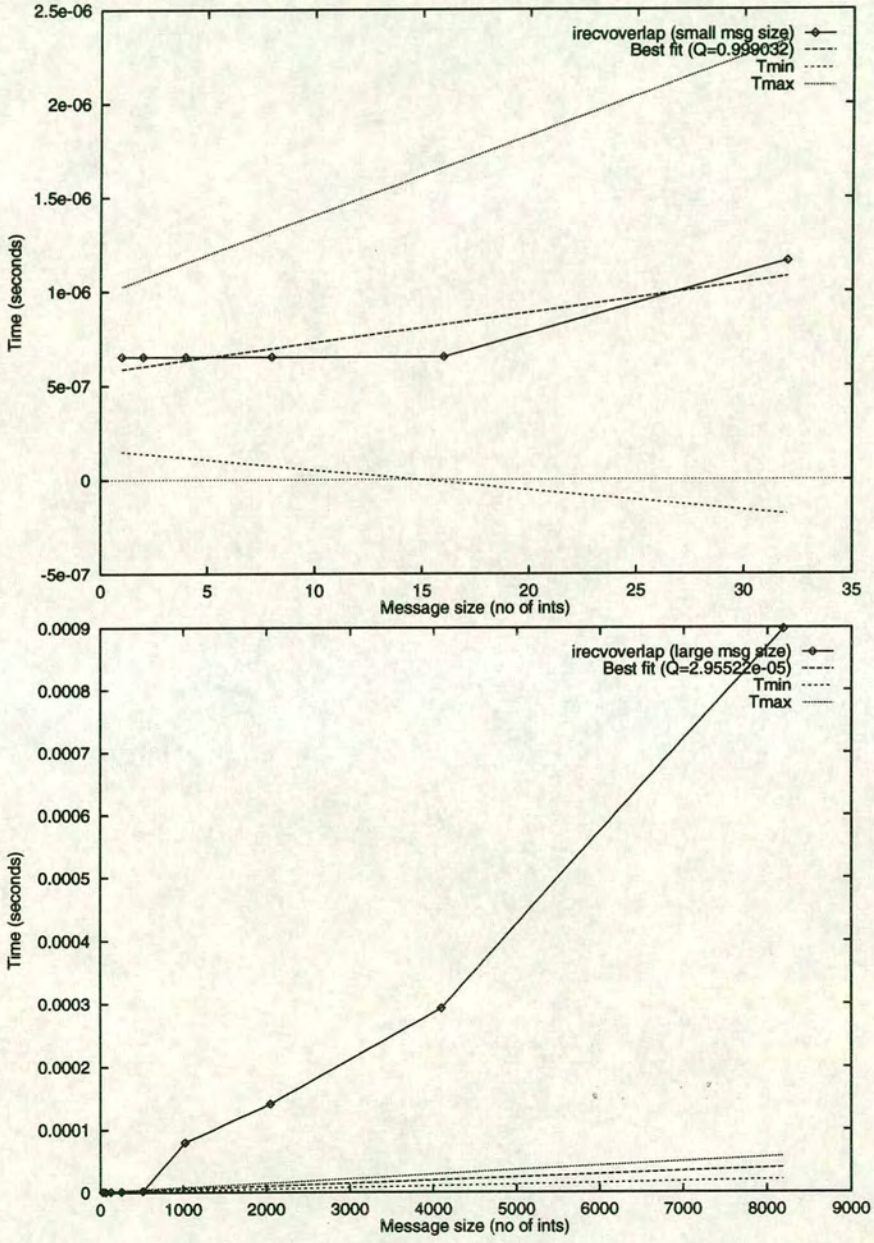
$$T_{irecv1}(\mu s) = \begin{cases} (30 \pm 8) + (0 \pm 1) \times ndata & \text{if } ndata <= 32 \\ (40 \pm 8) + (0 \pm 0.002) \times ndata & \text{if } ndata > 32 \end{cases}$$

irecv2



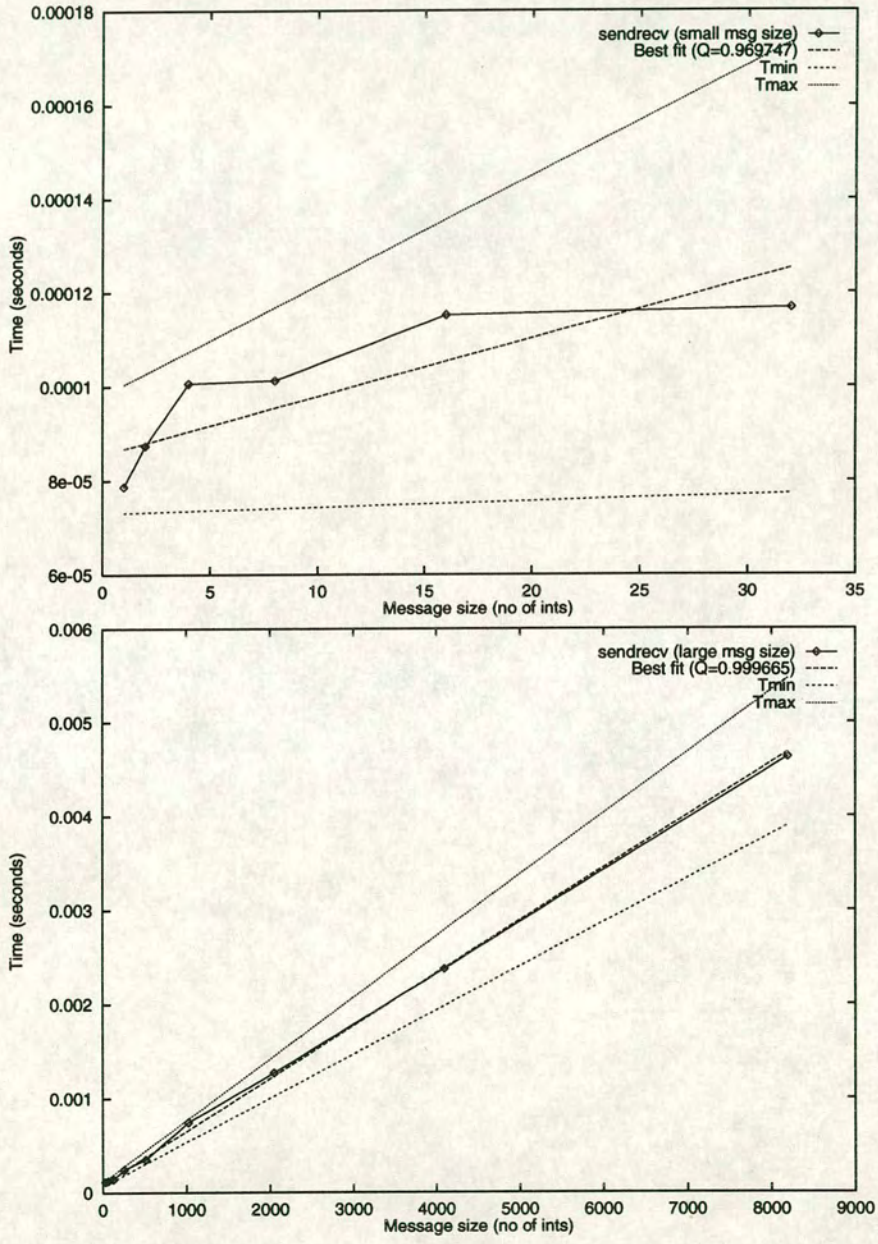
$$T_{irecv2}(\mu s) = \begin{cases} (40 \pm 7) + (0 \pm 0.5) \times ndata & \text{if } ndata <= 32 \\ (5 \pm 10) + (0.5 \pm 0.07) \times ndata & \text{if } ndata > 32 \end{cases}$$

irecvoverlap



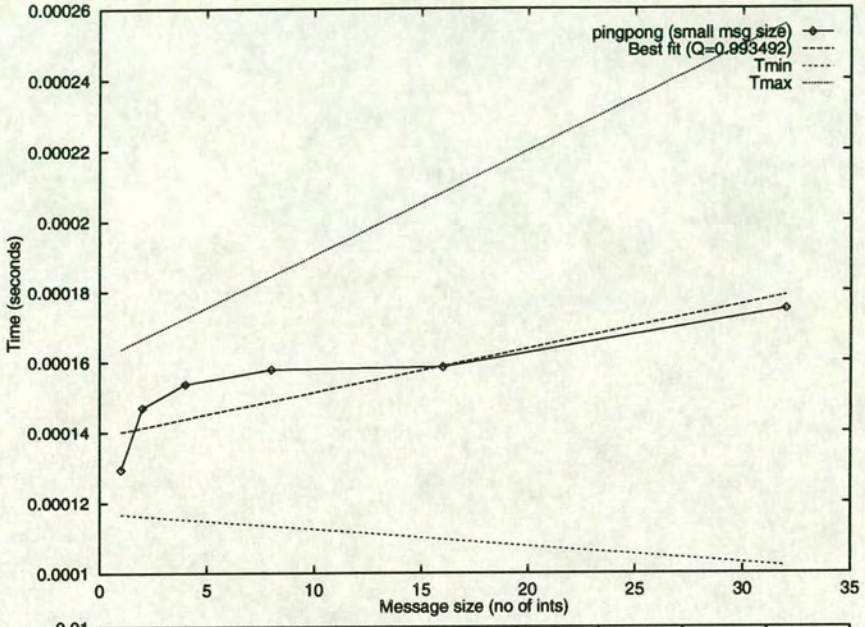
$$T_{irecvoverlap}(\mu s) = \begin{cases} (0.6 \pm 0.4) + (0.02 \pm 0.03) \times ndata & \text{if } ndata <= 32 \\ (0.5 \pm 0.5) + (0.005 \pm 0.002) \times ndata & \text{if } ndata > 32 \end{cases}$$

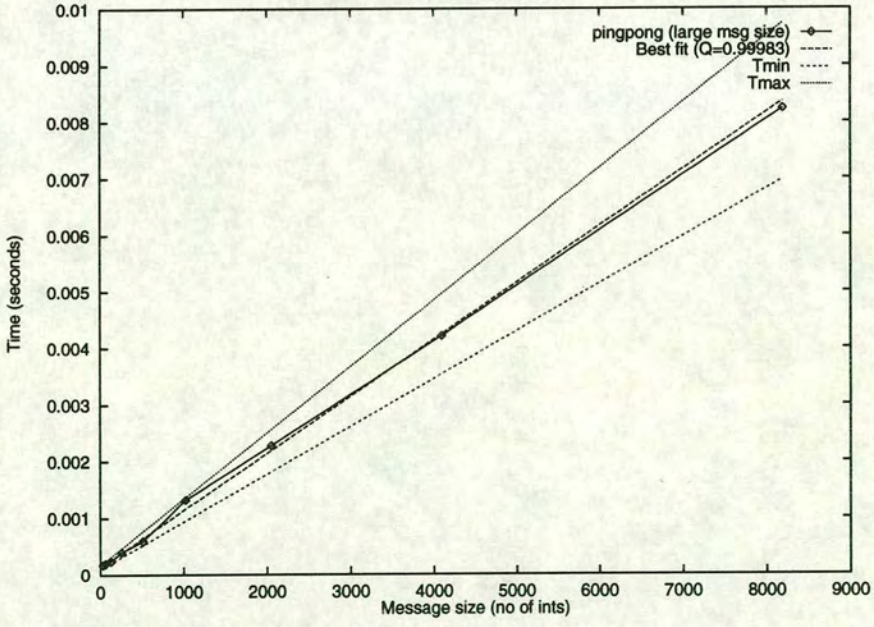
sendrecv



$$T_{sendrecv}(\mu s) = \begin{cases} (90 \pm 10) + (1 \pm 1) \times ndata & \text{if } ndata <= 32 \\ (90 \pm 20) + (0.6 \pm 0.09) \times ndata & \text{if } ndata > 32 \end{cases}$$

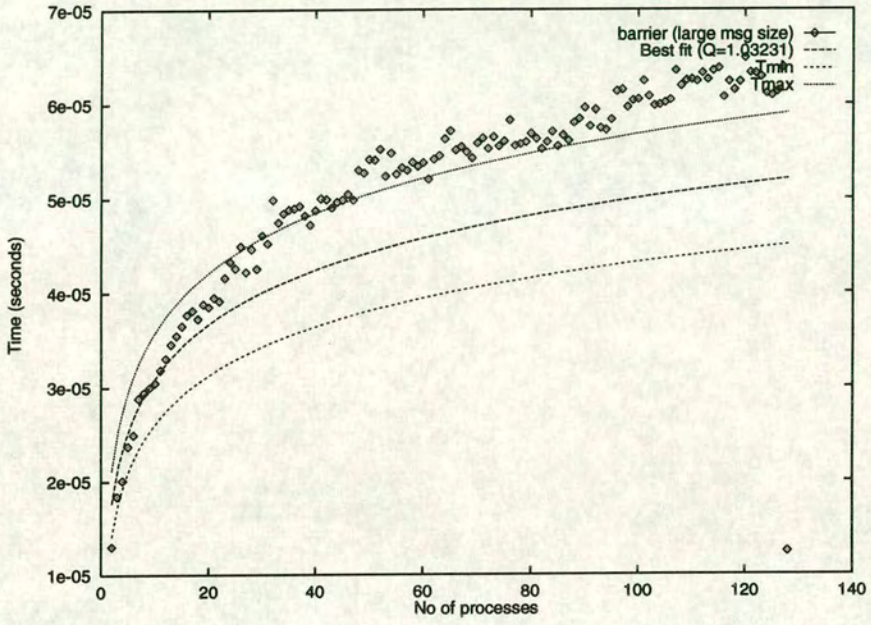
pingpong





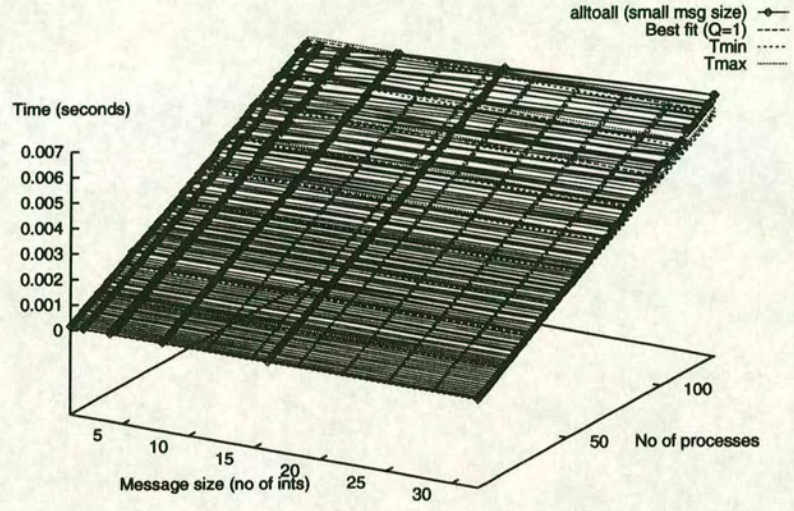
$$T_{pingpong}(\mu s) = \begin{cases} (100 \pm 20) + (1 \pm 2) \times ndata & \text{if } ndata <= 32 \\ (100 \pm 40) + (1 \pm 0.2) \times ndata & \text{if } ndata > 32 \end{cases}$$

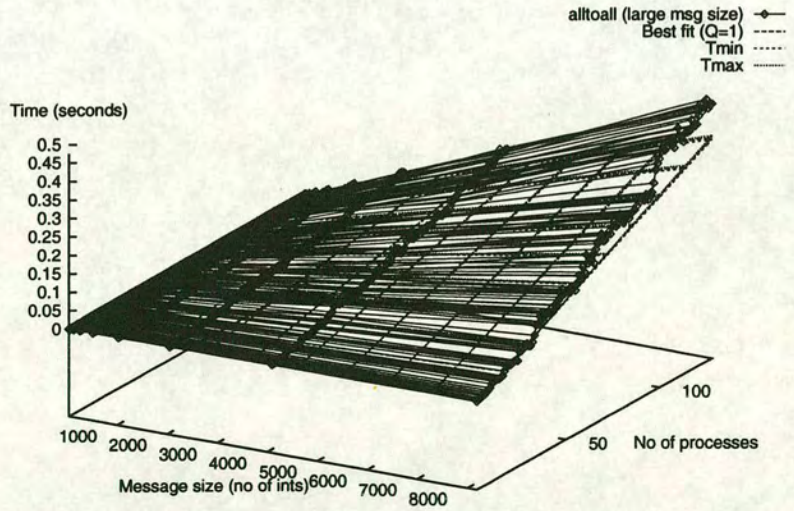
barrier



$$T_{barrier}(\mu s) = (10 \pm 3) + (8 \pm 0.8) \times log(nprocs)$$

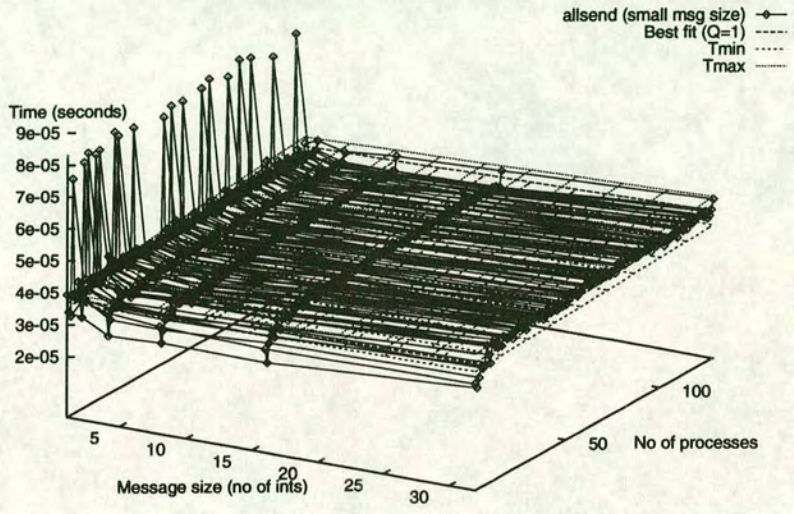
alltoall

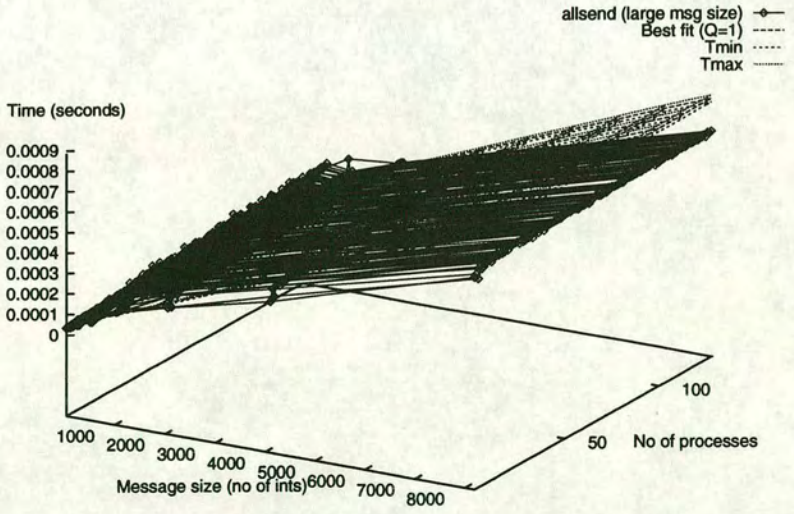




$$T_{alltoall}(\mu s) = \begin{cases} (40 \pm 20) + (50 \pm 0.7) \times nprocs + (2 \pm 1) \times ndata & \text{if } ndata <= 32 \\ (0 \pm 20) + (40 \pm 1) \times nprocs + (0.3 \pm 0.005) \times nprocs \times ndata & \text{if } ndata > 32 \end{cases}$$

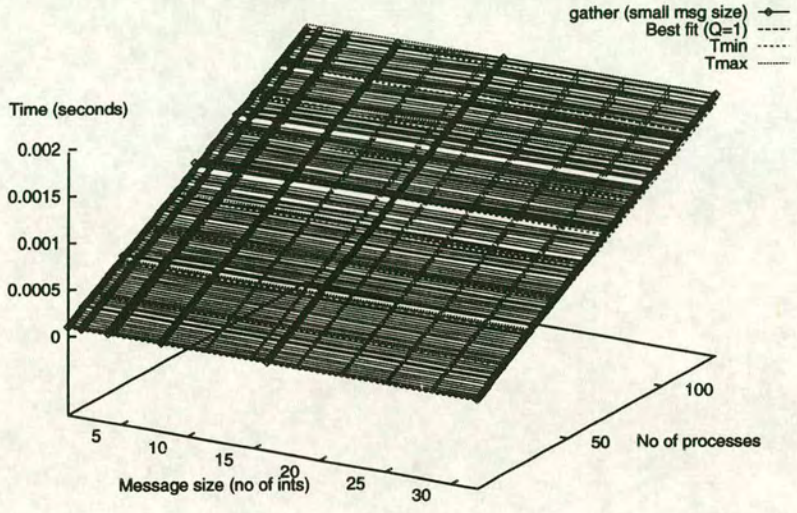
allsend

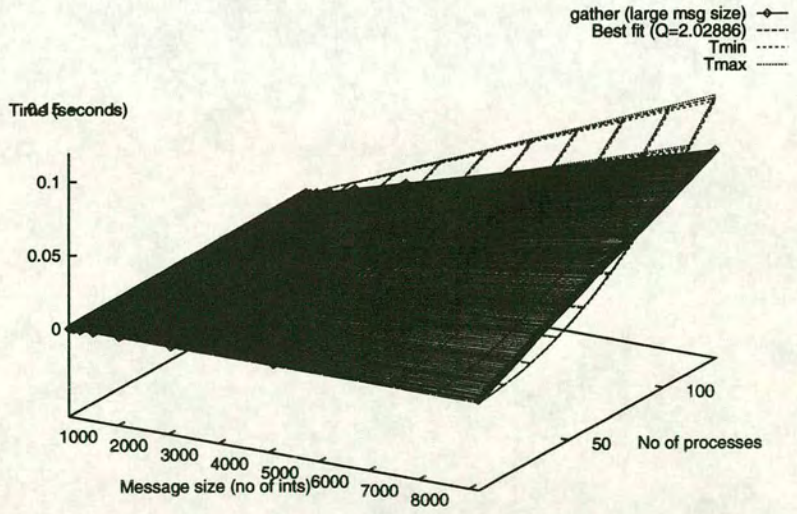




$$T_{allsend}(\mu s) = \begin{cases} (40 \pm 1) + (0.05 \pm 0.01) \times nprocs + (0.09 \pm 0.05) \times ndata & \text{if } ndata <= 32 \\ (40 \pm 1) + (0.1 \pm 0.02) \times nprocs + (0.1 \pm 0.002) \times ndata & \text{if } ndata > 32 \end{cases}$$

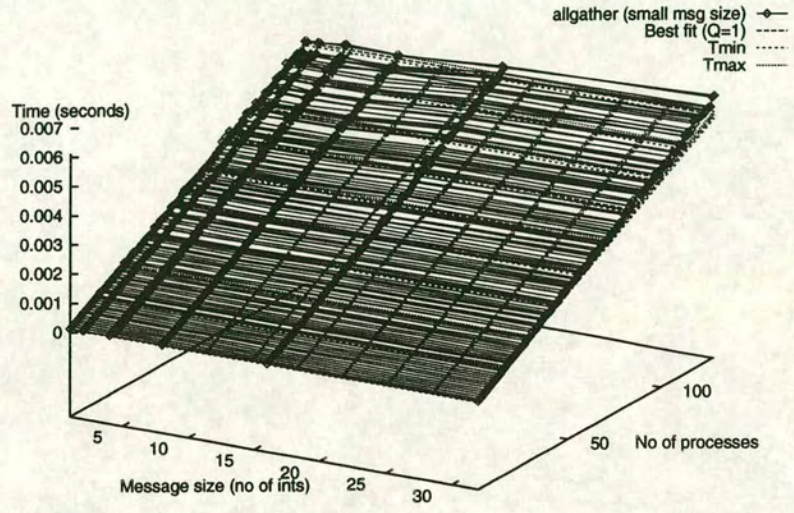
gather

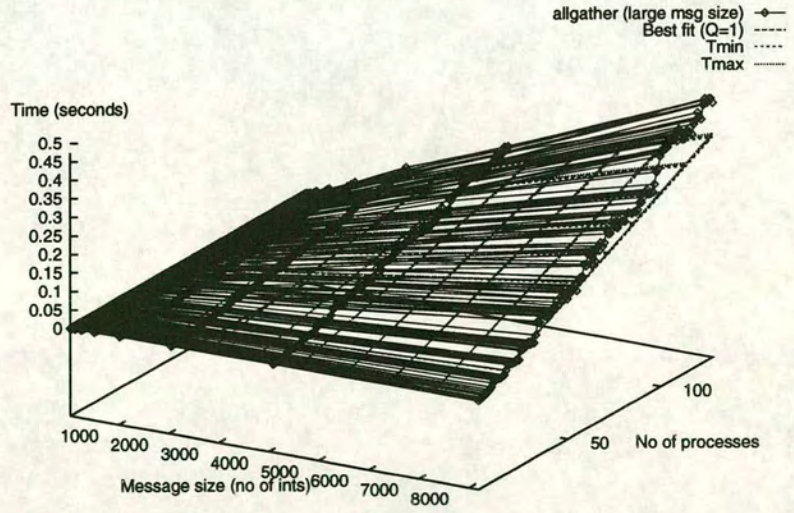




$$T_{gather}(\mu s) = \begin{cases} (70 \pm 8) + (10 \pm 0.2) \times nprocs + (0.7 \pm 0.5) \times ndata & \text{if } ndata <= 3 \\ (0 \pm 20) + (200 \pm 7) \times log(nprocs) + (0.0009 \pm 1e-05) \times nprocs^2 \times ndata & \text{if } ndata > 32 \end{cases}$$

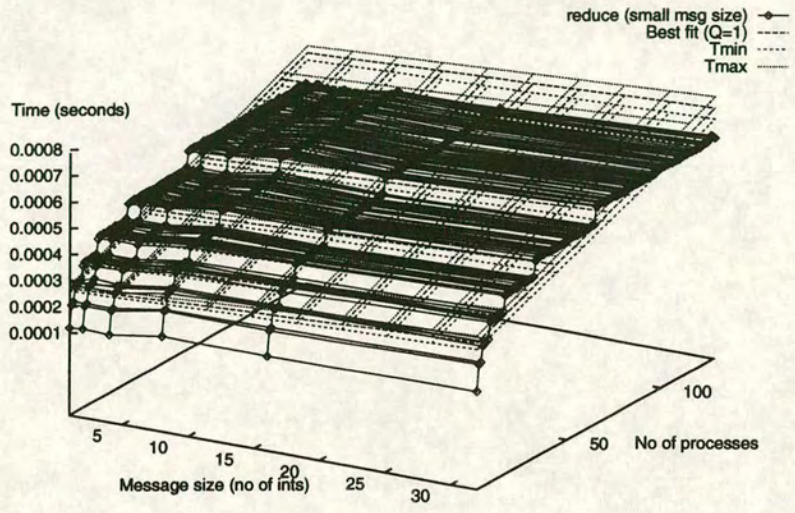
allgather

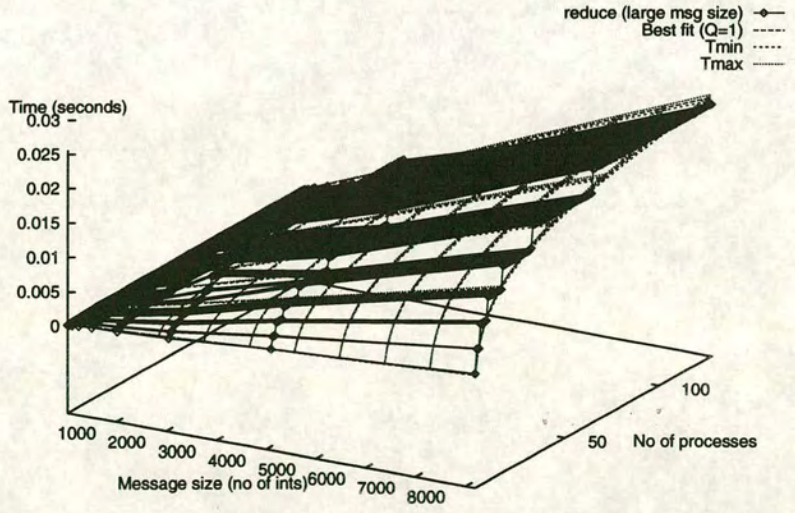




$$T_{allgather}(\mu s) = \begin{cases} (40 \pm 10) + (40 \pm 0.6) \times nprocs + (1 \pm 0.9) \times ndata & \text{if } ndata <= 32 \\ (0 \pm 20) + (40 \pm 1) \times nprocs + (0.3 \pm 0.005) \times nprocs \times ndata & \text{if } ndata > 32 \end{cases}$$

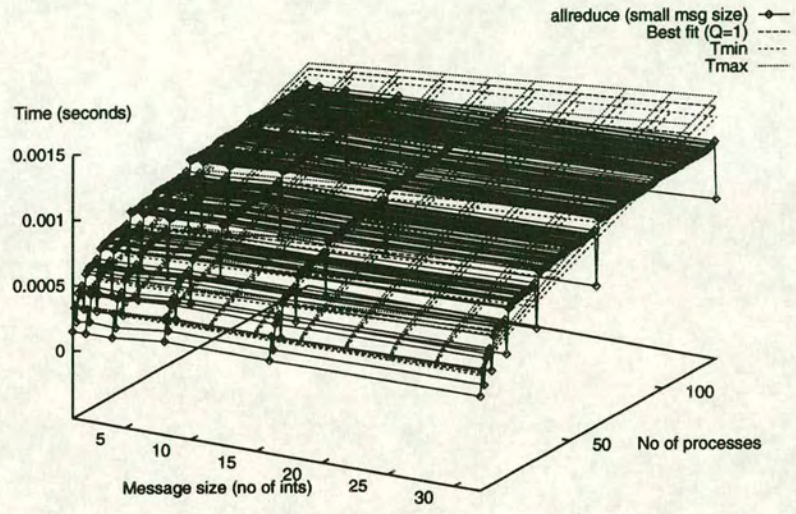
reduce

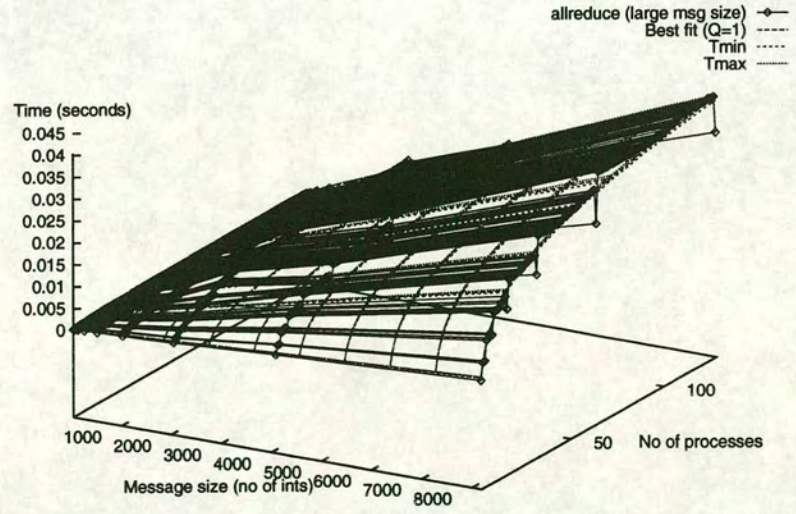




$$T_{reduce}(\mu s) = \begin{cases} (300 \pm 9) + (3 \pm 0.1) \times nprocs + (2 \pm 0.5) \times ndata & \text{if } ndata <= 32 \\ (300 \pm 10) + (2 \pm 0.2) \times nprocs + (0.6 \pm 0.01) \times log(nprocs) \times ndata & \text{if } ndata > 32 \end{cases}$$

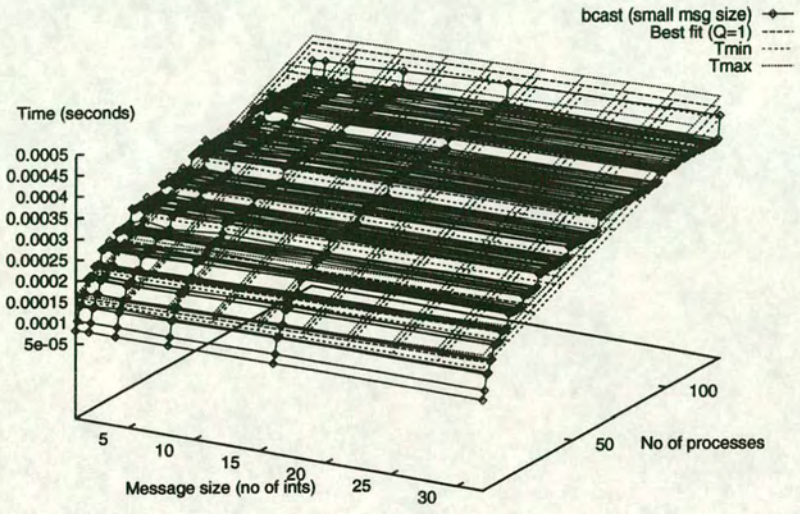
allreduce

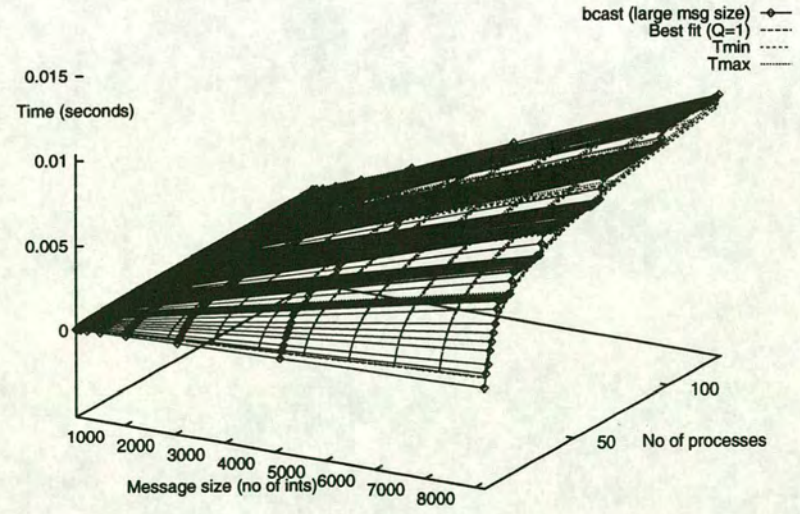




$$T_{allreduce}(\mu s) = \begin{cases} (300 \pm 10) + (6 \pm 0.2) \times nprocs + (2 \pm 0.2) \times log(nprocs) \times ndata & \text{if } ndata <= 32 \\ (300 \pm 20) + (6 \pm 0.3) \times nprocs + (1 \pm 0.01) \times log(nprocs) \times ndata & \text{if } ndata > 32 \end{cases}$$

bcast





$$T_{bcast}(\mu s) = \begin{cases} (200 \pm 6) + (2 \pm 0.08) \times nprocs + (0.6 \pm 0.3) \times ndata & \text{if } ndata <= 32 \\ (100 \pm 7) + (2 \pm 0.1) \times nprocs + (0.2 \pm 0.004) \times log(nprocs) \times ndata & \text{if } ndata > 32 \end{cases}$$

Appendix C

Papers

Copies of the papers related to the work in this thesis are included in this appendix. The references are:

F.W. Howell. **Reverse Profiling**. In I. Jelly and I. 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.

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.

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.

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.

Reverse profiling

F.W. Howell
University of Edinburgh
Department of Computer Science, J.C.M.B, The King's Buildings,
Mayfield Road, Edinburgh, EH9 3JZ, Scotland.
Telephone: +44 131 650 5141. email: fwh@dcs.ed.ac.uk

Abstract

This paper addresses the problem of designing parallel message passing programs with a reasonable idea of how well they will actually perform before they are run.

Models with very few parameters (e.g. LogP, PRAM) sacrifice accuracy to simplify design. By contrast, simulation techniques provide a good degree of accuracy by incorporating sophisticated architectural models, but present a "black box" to the user. This paper suggests a compromise between the two extremes, using an automatically generated model with a large number of parameters (a separate equation for each MPI function) which is presented to the user rather than being hidden within a black box. The profiling interface of MPI may be used "in reverse" to insert (rather than measure) expected timings from the model.

Keywords

MPI, profiling, performance prediction

1 INTRODUCTION

Programming parallel machines is somewhat of a black art as it is hard to know how well a program will run on a machine before actually running it.

The ideal model for designing parallel programs would be both simple to use and accurate in its predictions. However such a model doesn't yet exist, the simple models which are usable do not predict what actually happens reliably and the models which are fairly accurate (such as the simulation techniques) are both too cumbersome for general use and also present an opaque "black box" view of an architecture whose mysterious inner workings are not exposed. This leads to a development approach similar to the post mortem profiling technique used on actual machines.

The real challenge is to develop an approach which yields useful design information without requiring too much effort on the part of the programmer; if the method is too involved and complex then the programmer won't use it and will revert to post mortem tuning.

The technique of "reverse profiling" addresses some of these problems. There are two strands to the approach:

- The model is automatically generated by running an "MPI characterisation" routine on an architecture, rather than being crafted from in-depth knowledge of the architecture. The model is made available to the programmer for constructing quick pencil/paper analyses of performance.
- Since performing these calculations becomes tedious, especially when evaluating performance on a range of machines and problem sizes, a method is included for automatically computing these delays using the profiling interface of MPI. Rather than use profiling to *extract* timing data from a run of a program, "reverse profiling" *inserts* estimated times.

The performance model consists of separate equations for each MPI function giving the average, minimum and maximum times for a given number of processors and message size. These equations are generated automatically by an MPI program which times each MPI function with a range of message and group sizes, then fits an appropriate equation to the data. Running this on an architecture produces a LaTeXdocument with the equations for each function and graphs of the timing data used to generate the equations. This "datasheet" may be used by the programmer for quick estimates of the time an MPI function will take. A summary file is also produced for the reverse profiling.

The equations given in the model may be used for analytical performance predictions of a program, possibly in conjunction with a spreadsheet or graphing package to experiment with alternative designs at an early stage.

Alternatively the evaluations may be done by the computer using reverse profiling. This involves linking in an extra library, in exactly the same way as a normal profiling interface is linked. The reverse profiling library intercepts each call to an MPI function in the program, uses the appropriate equation to estimate the time the function would take and generates a trace file in a similar manner to a standard profiler. It then calls the normal MPI function to actually perform the communication.

The next section describes related techniques for performance prediction; section 3 describes the routines for generating the model of MPI performance and section 4 details reverse profiling. This is followed by an example and conclusion.

2 OTHER TECHNIQUES

Many approaches have been suggested to tackle the problem of performance prediction; the two ends of the spectrum are simple models like LogP (Culler, 1993) and detailed simulation (Brewer, 1993). Foster (1994) provides an interesting description of parallel design techniques. Driscoll (1995) uses an approach based on an extension of Amdahl's law to look at the performance of a program in terms of equations describing the sequential and parallel sections, a higher level view of performance prediction than the approach of this paper.

Getting closer to the source code level, Sarukkai (1994) addresses the problem of scalability analysis, using the SAGE/SIGMA toolkit to derive a program graph which is analysed to produce a complexity model. Wabnig (1995) also represents the program by a directed graph and the hardware by a processor graph, noting that these graphs get very large for real programs.

LAPSE (Dickens 1993) uses a parallel simulation technique for performance predictions

of message passing programs on the Intel Paragon. It uses a simple delay model for point to point communications and provides its own versions of the collective calls written in terms of these.

Reverse profiling is intended as a practical quick approach for the many programmers relying on post mortem techniques at present. It scores over other approaches in providing models directly based on the parallel primitives the programmer sees and in being as straightforward to use as standard profiling. It is not a revolutionary approach; rather a step towards the ideal of pre-natal design rather than post-mortem analysis of parallel programs.

3 GENERATING THE MODEL

It would be useful if performance models for MPI were supplied along with the libraries, but this is not the case, so they need to be generated. A model for point to point communication is not sufficient as much use is made of collective communication calls in MPI, such as `MPI_Bcast`, `MPI_Alltoall`, `MPI_Reduce`, `MPI_Barrier` etc. These all have different performance characteristics which are not adequately described by simple point to point models such as LogP. Parallel benchmarks tend to be directed towards comparing machines rather than providing design data for programmers.

Nupairoj (1995) describes an approach to benchmarking the MPI collective communications which attempts to work out how the structure of the underlying implementation of the collective MPI functions in order to derive reasonable performance models. In contrast the technique described below simply provides equations to *describe* the delays seen by a programmer calling each MPI function. A characterisation run only needs to be performed once for each architecture of interest to generate the required model.

3.1 Measuring performance of MPI building blocks

Characterising the performance of the MPI functions is straightforward in principle; measure the time to complete N calls and take the average. The parameters of interest are the number of processors and the size of the messages.

To time an operation (e.g. `MPI_Bcast()`), a short function is written:-

```
void time_Bcast(int numelems, double &time)
{
  int *buffer = new int[numelems];
  MPI_Barrier( comm );
  double e1 = MPI_Wtime();

  MPI_Bcast( buffer, numelems, MPI_INT, 0, comm );

  time = MPI_Wtime() - e1;

  time = getmax( time );
  delete buffer;
}
```

The `MPI_Wtime()` function is used to time the operation. The processes are synchronised beforehand using an `MPI_Barrier`. This is not perfect, as some processes may return from the barrier before others, so an alternative synchronisation technique has also been used which first determines the clock skew between different processes' `MPI_Wtime()` values, then busy waits until the timer reaches an agreed value. This provides synchronisation to a resolution of the short time required to read the timer, but just using `MPI_Barrier` is more convenient in practice.

The time is measured from this synchronisation point until the last process has returned. The `getmax()` function uses an `MPI_Reduce` across all processes to determine this maximum delay.

The parameters are the size of the message and the number of processes in the current communication group `comm`. These are varied across the range of values of interest on the machine, and each timing is repeated to produce a 3D set of measured times of the operation on the machine.

A surface is then fitted to this data using a least squares technique. It is not known beforehand what form the equation should take. There may be a constant start up cost with a linear data dependent factor for the message to be transferred across the network; or a data dependent startup (corresponding to an initial copy of the message into an internal buffer) with a data independent transfer cost (in a shared memory machine); the time may grow linearly with the number of processors, or with the logarithm of the number of processors for tree based algorithms; there may well be a network contention factor which predominates with large messages. The list of possible factors is endless and varies from machine to machine and from MPI function to MPI function.

Determining all the physical machine and algorithm parameters is not the aim of this approach. The aim is a descriptive equation which is simple enough to use and which provides confidence intervals to indicate the goodness of the fit. No claim is made that the parameters correspond directly to anything in real life; the only claim is that they fit the measured data to a given degree of accuracy.

In order to obtain this elusive compromise between a simple equation and an accurate fit, a brute force approach is taken performing a range of different curve fits and selecting and the best. The equations for the time of an operation in terms of the number of processes in the group p and the message size d take the form of a constant factor, a "startup parameter" dependent on the number of processors, and a "data dependent" factor dependent on the message size and the number of processors:-

$$t(p,d) = \text{c\_coeff} + \text{s\_coeff} * \mathbf{startupfn(p)} + \text{d\_coeff} * \mathbf{datafn(p,d)}$$

$$\mathbf{startupfn(p)} = \text{one of} \left\{ \begin{array}{l} p \\ \log(p) \\ p^2 \end{array} \right.$$

$$\mathbf{datafn(p,d)} = \text{one of} \left\{ \begin{array}{l} d \\ pd \\ \log(p)d \\ p^2 d \end{array} \right.$$

Thus a total of 12 curve fits are performed using every combination of the startup and

data functions. These functions were chosen as they provide reasonable fits for all cases thus far encountered. It was originally hoped to provide an adequate fit using one or two coefficients but this wasn't sufficient for the collective calls.

A fit is performed to determine the three coefficients using all combinations of the two functions and the one with the minimum chi-squared value is selected. Estimates of the standard error of each coefficient are also produced. These yield equations giving the maximum and minimum expected times. This should only be used as a rough guide, as there is no guarantee (or even likelihood) that the measured data conforms to a normal distribution. However, it is useful to have at least some indication of expected confidence intervals.

An example equation for `MPI_Allreduce` is:-

$$T_{allreduce}(\mu s) = \begin{cases} (50 \pm 30) + (200 \pm 10) \times log(p) + (4 \pm 1) \times d & \text{if } d <= 32 \\ (300 \pm 30) + (20 \pm 2) \times p + (0.9 \pm 0.03) \times log(p) \times d & \text{if } d > 32 \end{cases}$$

Separate equations are given for "small" and "large" messages as the shape of the fit often differs.

3.2 Output formats

The model is intended to be available for programmers to have an idea of the delay imposed by each MPI function. Because of this, one of the output formats is an automatically generated LaTeXdocument listing the equations and giving graphs of both the raw data and the fitted surfaces. Figure 1 gives an example page from a datasheet. The other output format is a summary file for computer based tools (such as the reverse profiler) to read.

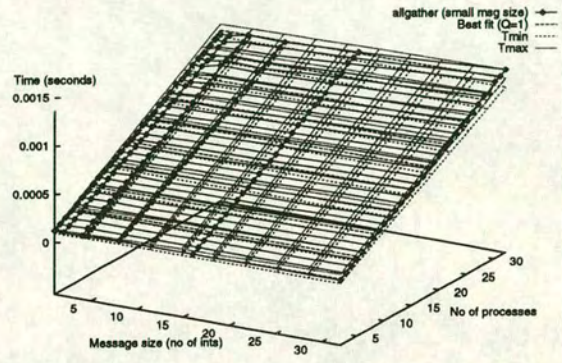
4 REVERSE PROFILING

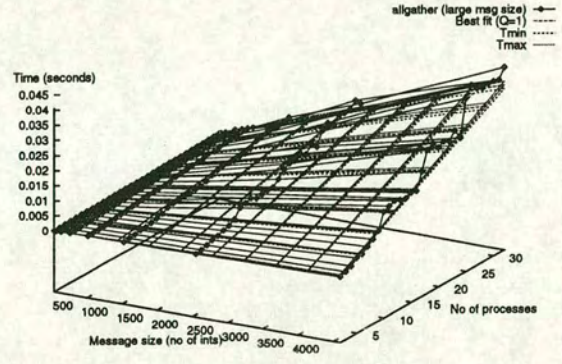
Reverse profiling is a technique which applies the MPI performance model for an architecture to a user's program to generate an estimate of the run time on that architecture. It uses the MPI profiling interface to intercept the user's calls to MPI functions and calculate the expected delay before returning control to the MPI routine to do the actual work.

Each process keeps track of its own simulation time and updates it whenever an MPI function is called. This means a normal trace file can be generated. A model of any machine may be used, and any MPI implementation can be used as the development environment. For example, workstation implementation of MPI may be used with a Cray T3D model to generate predictions of performance on the parallel machine.

Because it does not involve full simulation, it can't be applied to non-deterministic routines, for example those employing dynamic load balancing. However, the performance model will provide the key design data for such routines (such as the minimum and maximum message times). For non-deterministic programs the method must be combined with pencil and paper calculations, or with times measured from the target machine. Note that non-deterministic programs are likely to strain simulators and profilers too, since a minor miscalculation of delay may affect the outcome. A large proportion of useful parallel

allgather





$$T_{allgather}(\mu s) = \begin{cases} (50 \pm 20) + (40 \pm 1) \times nprocs + (1 \pm 0.9) \times ndata & \text{if } ndata <= 32 \\ (4 \pm 20) + (40 \pm 3) \times nprocs + (0.3 \pm 0.009) \times nprocs \times ndata & \text{if } ndata > 32 \end{cases}$$

Figure 1 A page from an automatically generated MPI data sheet.

programs are deterministic. Reverse profiling is a simple usable technique aimed at the majority of programs.

4.1 Results generated using reverse profiling

Running a reverse profiled MPI program produces a trace file which may be displayed as a timing diagram. Repeated runs may be used to produce graphs showing how performance varies with the problem size and number of processors in the machine. The machine model is supplied at run time as an environment variable pointing to a file produced by the MPI characterisation routines.

4.2 The technique in detail

MPI (MPI Forum, 1995) provides a simple profiling interface; all the `MPI_` functions are also accessible with the prefix `PMPI_`. Profiling (or reverse profiling) code may be added by writing substitute `MPI_` functions which perform the necessary (reverse) profiling task and call the `PMPI_` function to do the actual work. The linker ensures that the appropriate functions are called. The compilation commands to compile a normal MPI program, to compile with a profiler and to compile with the reverse profiler are:-

```
cc prog.c -lmpi
cc prog.c -lprof -lpmpi -lmpi
cc prog.c -lrevprof -lpmpi -lmpi
```

Each process has a `double the_time` variable to store its current simulation time. The profiled versions of the MPI functions update `the_time` according to the performance equation for that function and write lines to the trace file.

For point-to-point communications the receiver needs to know the time the sender started sending the message in order to work out when it should arrive. The minimum delay at the receiving end occurs when the message has been posted by the sender well in advance and the message has only to be copied from a system buffer. If the `send` starts at the same time as the `recv`, there will receiver will suffer an additional wait time for the message to arrive. This will be worse if the sender starts after the receive does.

For collective operations involving synchronisation (i.e. the majority of them), each process must know the start time of every other. Thus a point-point reverse profile function looks like:

```
int MPI_Send( data, dest, ...)
{
  // Send the_time to the destination
  PMPI_Send(the_time, dest, ...);
  the_time += /* computed delay for the message */;

  // Perform the actual send
  PMPI_Send( data, dest, ... );
}

int MPI_Recv( ... )
```

```
{
  // Recv the sender's start time
  // Compute the recv delay the_time
  // function of ( the_time, sender_start, message size )
}
```

and a collective operation:–

```
int MPI_Barrier()
{
  // MPI_Allgather to get each process's the_time
  // Set local the_time to the latest of all the_times
  // Plus the computed delay for the barrier.
}
```

This works as long as two conditions are met:

1. MPI_Recv is not allowed wildcarded receives. This is because there are two receives (one for the sender time, one for the actual data) which couldn't be guaranteed to come from the same source. This problem is related to the non-determinism issue raised earlier. A solution would be to tag the timestamp onto the main body of the message, or to do a wildcarded receive for the first message, work out where it came from, and do a receive from there.
2. Collective operations imply synchronisation.

At present a trace file is generated which may be displayed with the HASE timing diagram tool (Howell, 1994). Additional tracing (e.g. source code line numbers) could be added if necessary. Each process generates a separate trace file (`p<rank>.trace`), and repeated runs may be combined to produce scalability graphs.

4.3 Estimating the computation delays

The reverse profiling technique has accounted for the communication costs quite happily, but the times for user code have not been accounted for. Even without considering compute times, useful results may be obtained since the amount of time spent in idle "wait" states can be measured from the timing diagram and the communications structure of the code is clearly visible. None of the techniques thus far encountered by the author for this purpose are entirely satisfactory. In practice a combination of the following techniques for estimating computation time are used, with option 2 yielding the preferred tradeoff between hassle and accuracy:-

1. Fix it at 0. This is the mirror of the PRAM model which sets the computation cost at 1 and makes communication cost 0!
2. Let the user estimate it (in units of seconds, or number of memory accesses, arithmetic operations, etc).
3. Cycle count the assembly code.
4. Measure the times on the fly. This is only appropriate when developing on the target platform and not multitasking or multithreading on a single processor.

5. Measure the important times with a profiler off line.

Option 1, ignoring computation altogether, yields graphs showing the total communication time for an algorithm on a machine, which may be useful in itself as it shows how computation time must fall in order to make use of the machine. Option 2 is surprisingly useful. The programmer adds calls to a "`compute(N)`" macro which adds N "time steps" to the local simulated time, where a "time step" is the time taken to perform an arithmetic operation. This time is highly variable because of the influence of the memory hierarchy, but may be bracketed between likely limits (e.g. between 1 and 10 microseconds). This time step can be given as a parameter to the reverse profiler, so one may check how a design fares when given minimum expected compute step time and maximum expected communications time (the worst case for parallel algorithm scalability). Saavedra-Barrera (1989) describes characterisation routines for measuring the performance of different classes of operations in Fortran and if such figures were generally available for sequential code it would make parallel design easier.

Cycle counting of assembler code (option 3) is the preferred choice of the simulators. This technique has been shown to yield very accurate time estimates (Brewer, 1991). It involves an extra compilation stage, with the assembly code for the application being interpreted and augmented by a routine which inserts instructions to update a global cycle count after each basic block. Since the number of cache misses may lead to an order of magnitude variation in the execution time, a cache model is required for such simulators. This technique also requires augmented versions of all libraries used.

Experience using the Proteus `augment` tool indicated that though the technique works, it is too time consuming and awkward for quick estimates of compute time. It is also a "black box" approach and it it hard to know how reliable the estimates will be.

Option 4, measuring the compute times on the fly, is tricky on a multi-tasking system. Some multi-threading libraries provide "virtual timers" which only measure compute time consumed by the current thread, but these are not generally available. In any case, the compute times would have to be scaled for the target architecture.

The final option, profiling important subroutines on the target system and feeding the numbers back into the reverse profiler yields the most believable numbers.

5 EXAMPLE

This section illustrates results obtained by using reverse profiling with the `outer` routine from the Cowichan suite of problems (Wilson, 1994).

`outer` is given a set of N (x, y) coordinates and computes the distance of each point from every other point. These distances are stored in a $N \times N$ matrix. Since the distance from point A to point B is the same as from B to A, the matrix is symmetric about its diagonal. For N points, $N^2/2 - N$ distance computations are needed. The diagonal values of the matrix are all set to N times the maximum off-diagonal value. The routine also generates a real vector of distances of each point from the origin.

The MPI implementation of the routine generates the matrix and vector as distributed data structures, with an equal number of rows on each processor. Each process calls `MPI_Allgather` to take a local copy of the input points. It then computes the local section

of the vector and the matrix, performs an `MPI_Allreduce` to determine the maximum distance across the matrix and fills the local section of the diagonal.

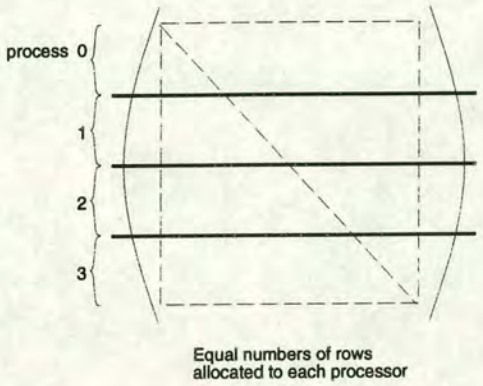


Figure 2 `outer` : matrix distribution across 4 processors

Each process computes the distances for all the matrix positions below the diagonal as well as those above it, thus doing twice the amount of work necessary, but not requiring any extra communication.

The routine is thus very simple, yet it is not trivial to work out how fast it will run on a range of problem and machine sizes.

A characterisation of the EPCC's implementation of MPI on the Cray T3D was generated using the routines described above. The `outer` routine was linked with the reverse profiling library on a workstation running the LAM implementation of MPI. The routine was then run on the workstation varying the number of processes and data sizes to obtain predictions of how it would perform on the T3D.

In the code, an example of one "compute step" is:

```
matrix[r - matrix.local_displ()][c] = d;
```

i.e. it is an extremely crude estimate of the time. A reasonable estimate of the time that this would take on the 150MHz DEC Alpha processors used in the Cray is hard to make without a detailed knowledge of the cache, compiler optimisations, pipelines and main memory latency. A direct execution simulator would work with the assembly code which enables the effect of compiler optimisations to be measured, but still leaves the pipelines and memory hierarchy to be modelled (which is possible, but not convenient).

The time for a basic compute step was left as a parameter and varied from $100ns$ up to $1us$ to see the effects on speedup, estimating that the line of code above (which includes a function call, a subtraction, two array indexing operations and a store to memory) would take between 15 and 150 cycles on a processor with a $6.6ns$ cycle time.

Figure 3 shows the measured and predicted speedups, which correspond reasonably with a compute step set between $0.1us$ and $1us$.

For this example reverse profiling gives a reasonable prediction of the speedup as long as the compute time can be estimated. It also allows "what if" experiments on a design to see how it can be expected to behave.

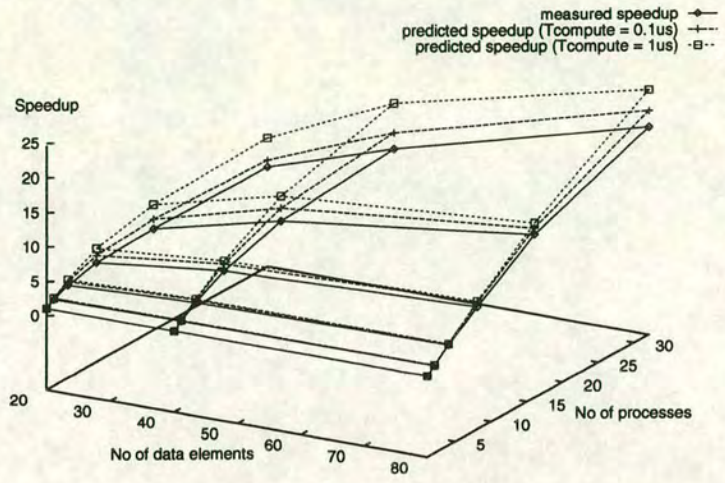


Figure 3 `outer` : predicted and measured speedups on the Cray T3D

6 CONCLUSIONS

Reverse profiling offers a very quick and easy method of performance prediction for MPI programs. Unlike simulation techniques it builds directly upon the full and complete MPI libraries available now. It doesn't attempt to handle non-determinism but this is the area in which existing profilers and simulators produce the least believable results. It works with any MPI implementation which provides the standard profiling interface, so predictions may be performed in parallel.

It is intended to complement rather than replace analytical approaches; making the model available to programmers allows pencil and paper analysis where appropriate.

The most important next stage is to obtain feedback from users to judge whether the current balance between simplicity and accuracy is appropriate. Work is also currently in progress investigating whether a similar technique could be applied to a shared memory programming model.

7 ACKNOWLEDGEMENTS

Thanks to Marcus Marr for suggestions on the MPI characterisation routines and also to the anonymous reviewers for their detailed and constructive comments.

REFERENCES

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

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

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R. and von Eicken, T. (1993) LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* San Diego, CA, May 1993.

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

Driscoll, M.A. and Daasch, W.R. (1995) Accurate predictions of parallel program execution time. *Journal of Parallel and Distributed Computing*, 25(1).

Message Passing Interface Forum (1995) MPI: A Message Passing Interface. Technical report, University of Tennessee.

Foster, I. (1994) *Designing and Building Parallel Programs*, chapter 3. Addison-Wesley. Available online at http://www.mcs.anl.gov/dbpp/.

Howell, F.W., Williams, R. and Ibbett, R.N. (1994) 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.*

Nupairoj, N. and Ni, L.M. (1995) Benchmarking of multicast communication services. Technical Report MSU-CPS-ACS-103, Michigan State University.

Saavedra-Barrera, R.H., Smith, A.J. and Miya, E. (1989) Machine characterisation based on an abstract high-level language machine. *IEEE Trans. on Comp.*, 38(12), 1659–1679.

Sarukkai, S.R. (1994) Scalability analysis tools for SPMD message-passing parallel programs. In *MASCOTS '94: Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems.*

Wabnig, H. and Haring, G. (1995) Performance prediction of parallel systems with scalable specifications - methodology and case study. *Performance Evaluation Review*, 22(2).

Wilson, G.V. (1994) Assessing the Usability of Parallel Programming Systems: The Cowichan Problems. In *Proceedings of the IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems.* Birkhäuser Verlag AG, April 1994.

8 BIOGRAPHY

Fred Howell received his BSc and MEng degrees in Microelectronic Systems Engineering from the University of Manchester Institute of Science and Technology in 1992. He is currently a PhD student at the University of Edinburgh Department of Computer Science where his research interests include the design of parallel hardware and software. He has been funded by EPSRC and by Digital (Scotland) Ltd.

Hierarchical Architecture Design and Simulation Environment

F.W. Howell, R. Williams, R.N. Ibbett
Dept of Computer Science
University of Edinburgh
Edinburgh, Scotland, EH9 3JZ
email: fwh@dcs.ed.ac.uk

Abstract

The Hierarchical Architecture Design and Simulation Environment (HASE) is a tool for modelling and simulating computer architectures.

Using HASE, designers can create and explore architectural designs at different levels of abstraction through a graphical interface based on X-Windows/Motif and can view the results of the simulation through animation of the design drawings.

1 The Motivation

Advanced simulation tools are available for low level electronic design, such as Spice for analogue circuits, and VLSI layout tools. However, tools for rapid prototyping of architectural ideas are few and far between. Simulation languages (such as SIMULA, SIMSCRIPT, DEMOS etc) can be used to model computer architectures, but the user has to be an expert on simulation. This is also the problem of general purpose simulation tools (e.g. SES/Workbench), where icons represent 'queues', 'servers' etc., and the link between a queueing model of an architecture and the architecture itself is not immediately apparent to the engineer not fluent in queueing theory.

Conventional languages (C, C++, occam) are often used to construct simulators, but this approach involves starting from scratch for each new project. User interface aspects are often neglected as the tool will be thrown away with the next architecture. This is very wasteful, as many aspects of computers are constant between different architectures. The object oriented approach offers a solution. Standard components (such as memories, microprocessors and interconnection networks) can be held in a library. They can be constructed and linked together graphically on screen to create a simulation of an architecture, in much the same way that standard components can be wired together in a semi-custom VLSI tool. The difference is that the simulation is not fixed to low level wires and chip pins, but is free to choose the appropriate abstraction level.

HASE addresses one of the four "Grand Challenges in Computer Architecture" identified by the Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing [1]:

> *... to develop sufficient infrastructure to allow rapid prototyping of hardware ideas and the associated software in a way that permits realistic evaluation.*

Any tool which attempts to bridge the hardware/software gap must provide models which are as understandable to the compiler writers working on instruction scheduling as they are to the computer engineers. Hierarchy is a fundamental concept in managing complexity for both hardware and software, so the simulation tool should be hierarchical to allow detailed simulations in areas of interest, and higher levels for the other parts of the architecture. This also helps to balance accuracy with fast simulation time. Computer engineering is concerned with cost as much as performance, so simulation models should be able to incorporate cost and other factors such as power consumption.

Complexity of parallel systems has led to the development of "performance tuning" as a step in parallel program development. However, we believe that such "post-mortem" analysis is too late, and that "pre-natal" parallel program design based on a simulation modelling tool is a better solution. The "right first time" approach of VLSI design (and Total Quality Management) should be extended to cover parallel software.

Some simulation tools employ "direct execution" to evaluate performance of parallel programs on parallel architectures (e.g. MIT/Proteus [11], Stan-

ford/Tango [12], WWT [13]). This is an excellent approach for obtaining fast, realistic simulations if the processor you're running the simulation on is very similar to the processor used in the parallel machine - but we believe that this assumption is sometimes too restrictive and a hierarchical tool can find wider application.

2 The Tool

The HASE tool is based on SIM++, a discrete event simulation language built as an extension to C++. SIM++ is used to describe the behaviour of basic components of a simulation. The user can link together these components 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.

HASE is more flexible than many parallel architecture simulators, and can model any parallel system, not just a restricted variety of MIMD/SIMD machines. The use of SIM++ allows high performance distributed simulation on networks, using a Timewarp algorithm based on Virtual Time [10], [5]. The hierarchy also aids development of fast simulations, as typically only some of the components need to be simulated at a low level. The object-oriented simulation language makes it easy to construct reusable components, and a library tool lets users share components with each other.

Once a simulation has been constructed, users need the answers to many performance questions quickly. HASE incorporates automatic graphical results analysis, and the hierarchy is used to get detailed or global performance summaries. The design can be animated at any level, with the state of each component changing on screen, and data packets moving down links. This can highlight bottlenecks, and is an excellent presentation aid. Design is an iterative process, so once changes have been made the simulation can be run again rapidly. A hierarchical timing diagram display of the states of each component can also be displayed, and logic-analyser style measurements can be taken. There are also facilities for overall performance statistics such as total cost, average performance, and power dissipation.

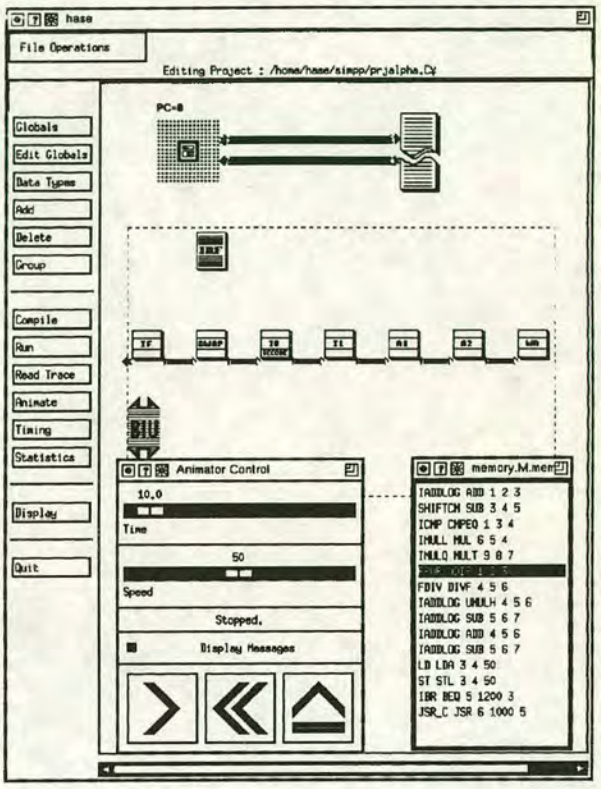


Figure 1: The HASE user interface.

HASE incorporates significant support for simulating single processors. Once a new assembly code has been defined, a parser is automatically linked in with the simulation, so that "instruction interpreting" objects can be written very easily. All "memory objects" are initialised at simulation run time with the appropriate program, fully parsed. These programs can be edited on screen, and whilst the simulation is running, the currently executing instruction can be highlighted. At higher levels, the overhead of interpreting instructions can be too great, so programs can be modelled as delays interspersed with communications events. Again parsers for the higher level language (for example consisting of `COMPUTE <time>`, `SEND <proc#>`, `RECV <proc#>`) can be generated automatically.

Simulation is particularly useful for parallel computers. Replication is a key feature of parallel systems, so HASE includes templates for the common structures. The user can slot any component into the template (including hierarchical components), and HASE does the rest. Current templates include a lin-

ear array, a 2D mesh, and an omega network (figure 2). More multistage networks and general k-ary n-cubes are under development. All have their size as a parameter, which can be set by moving a slider widget. Connections between neighbouring processors can be specified.

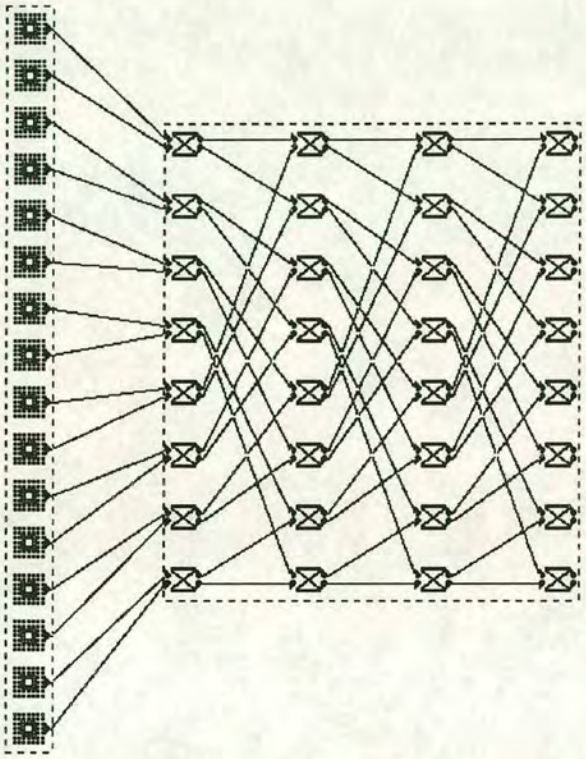


Figure 2: A HASE omega network.

In addition to simulating architectures, HASE can be applied to simulating parallel algorithms. The graphical interface and results analysis built into HASE can be used to visualise and evaluate parallel algorithms. Work has been done on process simulation for message passing systems, based on the Edinburgh Parallel Computer Centre (EPCC) CHIMP interface for portable message passing algorithms [9]. Work has also been done on routing algorithms for meshes and on evaluating multiprocessor WAN routers and LAN bridges [3].

3 Applications

3.1 DEC Alpha 21064 AXP microprocessor

1. At the detailed pipeline level, showing the movement of instructions through the functional units, blocking and exceptions (figure 1).

2. At the next level up, the instruction level. This provides a working 'instruction interpretting' simulation, with times for each instruction in clock cycles extracted from data sheets.

3. At the PMS level. Instructions are no longer interpreted - instead times for each process are modelled by delays, with measurements of total instruction counts, cache/main memory references, and I/O.

3.2 DEC Alpha-PC AXP

This demonstrates the use of HASE to evaluate complete systems. Cost/performance charts can be obtained. Note that the processor in this full system simulation is the one produced above, illustrating the reuse of components. A different processor can be substituted (e.g. Pentium) for performance comparisons.

3.3 Parallel Systems

This illustrates the HASE facilities for simulating parallel architectures. Processors and interconnection networks can be swapped in and out and performance of the resulting system can be measured.

Various direct and indirect networks are used to connect the processing elements. These include omega and mesh structures. The effects of varying processor and interconnect performance on parallel algorithms can be measured.

3.4 Parallel Software

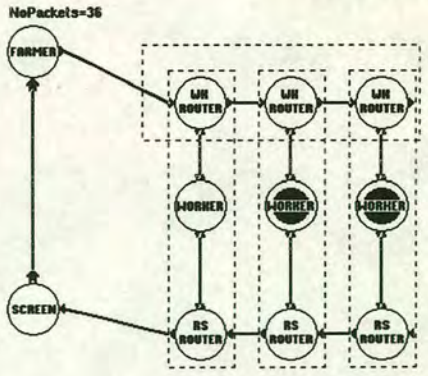


Figure 3: A task farm.

At the next level up are the applications. The demonstration shows the use of nested algorithmic skeletons [8] for developing parallel code.

4 HASE development

HASE has been developed by the Computer Systems Group at Edinburgh University. An advanced prototype using DEMOS was developed, and most of the ideas were tested on a multi level simulation of the Motorola 88110 microprocessor [2]. Recent work has converted HASE to the language SIM++ to allow faster simulations on networks of workstations. The current version of HASE runs on a Sun SPARCstation with X-windows and makes use of the commercial simulation language SIM++.

Currently work is in progress to include provision for evaluation of architectural support for PRAM and HPRAM computational models [6], and to evaluate process mapping strategies. Several PhD and MSc projects are using HASE as a simulation support tool. There are plans to make HASE freely available to the academic community.

References

[1] H.J. Siegel, S. Abraham, *et al "Report of the Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing"*, Journal of Parallel and Distributed Computing, 16, pp. 199-211, 1992.

[2] A.R. Robertson & R.N. Ibbett *"Simulation of the MC88000 Microprocessor System on a Transputer Network"*, Lecture Notes in Computer Science, Distributed Memory Computing, 2nd European Conference, EDMCC2, April 1991, Springer-Verlag.

[3] J.S. Westerman *"Parallel Processing for a Communications Node"*, M.Sc Dissertation, Univ. of Edinburgh Dept. of Computer Science, September 1993

[4] J. Birtwistle *"DEMOS: Discrete Event Modelling On Simula"*, Prentice-Hall, 1985

[5] D.R. Jefferson *"Virtual Time"* ACM Transactions on Programming, 7(3),(July 1985) pp404-425.

[6] T. Heywood & S. Ranka *"A Practical Hierarchical Model of Parallel Computation I: The Model"*, Journal of Parallel and Distributed Computing, 16, pp. 212-232, 1992.

[7] *"A Workbench for Computer Architects"*, IEEE Design & Test of Computers, Feb, 1988.

[8] M.I. Cole *"Algorithmic Skeletons: Structured Management of Parallel Computation"*, Pitman & MIT Press, 1989.

[9] J.G. Mills, Lyndon J. Clarke, Arthur S. Trew *"CHIMP Concepts"*, Technical Report EPCC-KTP-CHIMP-CONC, Edinburgh Parallel Computing Centre, April 1991.

[10] Jade Simulations Inc. *"SIM++ v3.8 Reference Manual"*.

[11] E.A. Brewer, C.N. Dellarocas, A.Colbrook, W.E.Weihl *"PROTEUS: A High Performance Parallel-Architecture Simulator"*, Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991

[12] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. *"Multiprocessor Simulation and Tracing Using Tango"*, Proc. 1991 ICPP, August 1991

[13] S.K. Reinhardt, M.D. Hill et al. *"The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers"*, Proc. 1993 ACM SIGMETRICS Conference, May 1993

HASE: A Flexible Toolset for Computer Architects

R. N. IBBETT, P. E. HEYWOOD AND F. W. HOWELL

Computer Systems Group, Department of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ UK

HASE is a Hierarchical computer Architecture design and Simulation Environment (HASE) which allows for the rapid development and exploration of computer architectures at multiple levels of abstraction, encompassing both hardware and software. The components of a computer system lend themselves naturally to being modelled as objects, so HASE has been implemented in an object-oriented language. Within HASE there are graphical entity design and edit facilities, entity library creation and retrieval mechanisms, an animator, and statistical analysis and experimentation tools for deriving system performance metrics. HASE uses an object-oriented database management system (ObjectStore) to make the design objects and the entity library persistent. For each architecture model HASE allows many experiments with varying parameters to be performed. The database facilities provided through HASE manage not only the results of each experiment, but also their relationship to the state of the architecture model that produced these results, including all input and output parameters and their values during the experiment. This paper describes the design of HASE, some of the varied projects which have used it, and the future direction of the system.

Received July 26, 1995, accepted November 8, 1995

1. INTRODUCTION

The Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing [1] identified four 'Grand Challenge Problems in Computer Architecture'. HASE, the Hierarchical computer Architecture design and Simulation Environment developed at the University of Edinburgh, is a tool which addresses the fourth of these: 'to develop sufficient infrastructure to allow rapid prototyping of hardware ideas and the associated software in a way that permits realistic evaluation'. Sophisticated VLSI design tools have been in existence for a number of years but it is only recently that attention has been focused on providing higher level simulation and animation tools for computer architects. Thus the HASE project has aimed to address two major problem areas: high level simulation and visualization of computer architectures, and simulation of parallel systems.

The hierarchical nature of computer architecture and design has been well understood for many years, e.g. Bell and Newell's PMS, ISP and RTL levels [2]. HASE allows the designer to move freely between these levels and to select the appropriate simulation level for different parts of the system in order to strike a balance between simulation accuracy and processing time. To meet all the aims for the environment, however, attention also had to be focused in the area between the domain of hardware simulators and general purpose simulation packages. Hardware simulators are typically inappropriate for dealing with software layers and general purpose simulation packages are not normally designed with hardware in mind. The usual approach to this problem is to write project-specific simulators in a language such as C++. This provides a high degree of flexibility, but also an amount of wheel re-invention.

Many commercial CAD tools are moving progressively towards higher levels of abstraction, and the use of hardware description languages such as VHDL and Verilog for hardware system simulation is becoming widespread. Since much effort has been invested in developing these toolsets it would be convenient to extend them to higher levels of simulation. However, most are not particularly suited to this task at present. In [3], for example, external C routines were written to compensate for VHDL's deficiencies in this respect.

Specialized tools include Ptolemy [4] at Berkeley which defines a framework for simulating and prototyping heterogeneous systems, and work at the University of Florida has involved simulating microprocessor-based parallel computers using processor libraries [5]. At UMIST the SES/workbench [6], a general queuing model tool, has been adapted to simulate the ARM processor [7]. At the Illinois Institute of Technology Chicago a prototype version of MIES [8] has been developed to visualize Register Transfer Level descriptions and a newer version is currently being implemented in an object oriented programming language.

At the same time, there has also been interest in developing mathematical formulations for modelling discrete event systems, most notably Zeigler's DEVS formalism [9] together with its primarily non-graphical implementation, DEVS-Scheme.

The ideas for HASE grew from a simulator built for an MC88000 system [10], written in occam and run on a Meiko Computing Surface at the Edinburgh Parallel

Computing Centre. However, since the components of a computer can be treated very naturally as objects, HASE itself has been developed using object oriented simulation languages, the first prototype [11] using DEMOS [12] and the current version Sim++ [13]. Sim++ is essentially a superset of C++ which includes a set of library routines to provide for process oriented discrete event simulation and a run time system for multi-threading many objects in parallel and keeping track of simulation time.

In the same vein, HASE now also uses an object-oriented database management system, ObjectStore [14]. The environment includes a design editor and object libraries appropriate to each level of abstraction in the hierarchy, plus instrumentation facilities to assist in the validation of the model. HASE also provides *model exploitation facilities* based on [15] and [16] allowing performance measurements to be derived from simulation runs. The system can thus be set up to return event traces and statistics which provide information at the PMS level, for example, about synchronization, communication and memory latencies.

The user interface to HASE is via an X-Windows/Motif graphical interface. Many complex systems of interacting components can be more easily understood as a picture rather than as words. In computer architecture the dynamic behaviour of systems is frequently of interest and HASE allows users to view the results of simulation runs through animation of the design window.

The first sections of this paper present an overview of HASE, the database organization and the HASE libraries. Then follows a description of the design of a system within HASE including test software to execute on the model architecture. This is followed by a description of the Sim++ code generated by HASE, and of the way a simulation is run. Later sections describe the various ways to view the results of a simulation, gather statistics and perform experimentation on the model architecture. Finally we present some of the projects which have used HASE and consider possible future developments.

2. OVERVIEW OF HASE

Figure 1 shows an overview of the HASE system. The core of each project undertaken using HASE is the *Architecture Description*. In the case of a multiprocessor, for example, this description consists of a collection of Processor, Memory and Interconnection Network *entities*. Each entity is a multi-faceted object having an instance name, an icon (usually a bitmap), a textual description, a list of its parameters, a list of ports and a pointer to its Sim++ simulation code. The design phase of a project involves selecting the appropriate entities from a *library*, or alternatively creating them, and linking them together to form the required system. Each entity's behaviour is described in the corresponding Sim++ method (the *body*). Once the design is complete the description is compiled to produce the simulation code for that system.

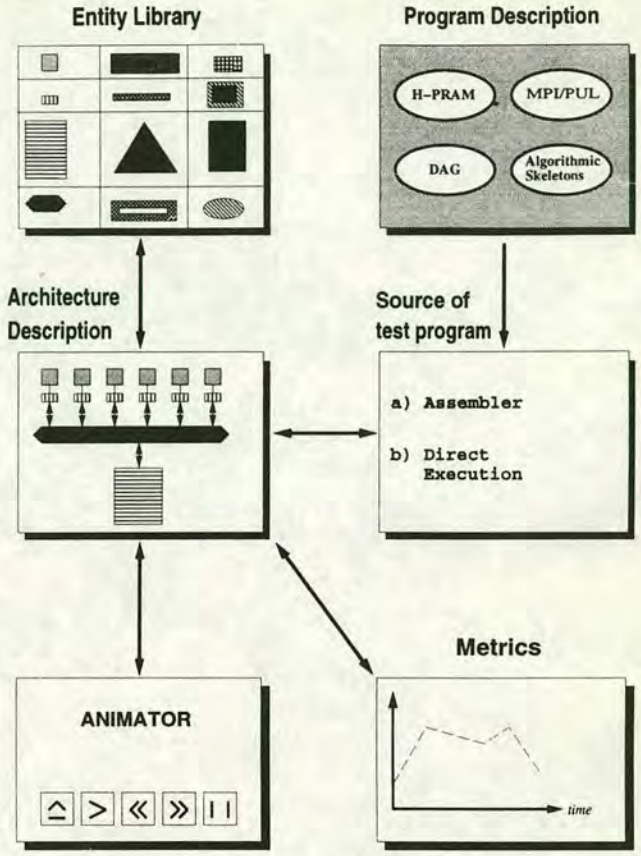


FIGURE 1. Overview of the HASE system.

To run a simulation, it is necessary to provide appropriate inputs. Thus test programs for the architecture can be written (in assembler, or any HLL with a compiler for the chosen processor), and the architecture, along with its program, can then be simulated. The output from the simulation run can then be used to animate the design, and thus provide visual feedback data to the designer, or to obtain performance metrics.

3. DATABASE ORGANIZATION

HASE includes an object-oriented database management system based on ObjectStore in order to allow all architecture design projects and the entity libraries to be maintained persistently. A major advantage of this approach is that in addition to its purely repository functionality, ObjectStore can also be used to manage the relationships and connectivity between objects. Furthermore, the use of object-oriented database technology provides the opportunity to exploit advanced transaction processing capabilities, such as nested transactions and rollback, and to facilitate the exploration of alternative paths while fine-tuning a model.

The database management system also allows versions of simulation models and experiments to be maintained so that an experimental program can proceed on an existing version of the model while subsequent versions

are under development. By integrating a C++-based object-oriented database management system with the existing HASE environment, most of the problems associated with impedance mismatch have been avoided. All HASE environment utilities are C++ based and all relevant classes are coherent throughout the environment.

Figure 2 shows how the databases are organized. The *user startup file* contains environment variable definitions for the default library databases and the user specific project directory containing subdirectories for all individual projects. Each user can have a number of project databases, each holding a number of projects, and a number of personal entity libraries in addition to having access to the public HASE entity library.

A project in HASE is interpreted to be the set of all entities, ports, links, parameters, etc., comprising the simulation model, together with their associated Sim++ code, bitmaps, etc., and the set of all experiments performed on the model. For each architecture model, a set of experiments may also be stored. Experiments typically involve changing the value of one of the parameters of a component of the architecture and running the same simulation for each parameter value. The database supports this experimentation facility by storing not only the results of each experiment, but also their relationship to the state of the architecture model that produced the results, including all input and output parameters and their values during the experiment.

4. LIBRARIES

Libraries in HASE are repositories for entities, the basic components of the Architecture Description. Each modular, reusable entity can be archived to a library for shared or later use or retrieved from a library for inclusion in an architectural model. The storage of pre-designed (and pre-tested) entities in the library means that HASE offers a reliable and convenient method for rapid prototyping.

The HASE Entity Library is a global read-only library, selected from possibly many shared libraries containing related entities. As a means of initially populating and supplementing this library for a specific site, entities from all projects migrate to a temporary holding area where the site database administrator determines which entities merit inclusion into a particular HASE Entity Library. The User Entity Library is a user's personal catalogue of entities. The number of libraries is virtually limitless, with the library in use defined as the most recently selected library.

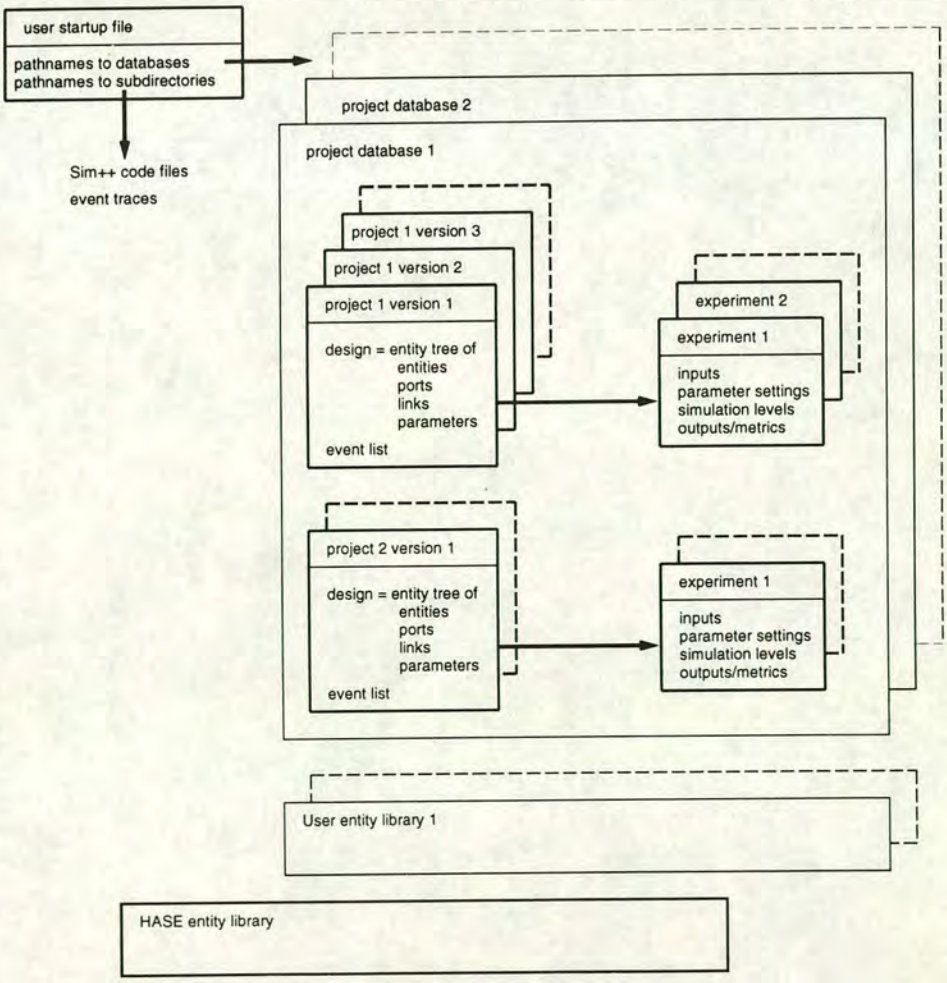


FIGURE 2. HASE database organization.

The design of a HASE Entity Library corresponds to the different levels of abstraction for the archived entities, e.g. PMS or RTL. An entity's hierarchy can be easily perused from within the library and can be included in the architectural model at various levels. The utilities are flexible enough to allow the user to map to other decomposition methodologies when creating the User Entity Library.

5. DESIGN

The basic constituent of the project is the Architecture Description which is a collection of entities with ports for data transmission across links to other entities or levels of decomposition. The architecture can be designed either *top down* by subdividing an entity into its constituent components or *bottom up* by grouping a set of components into a compound component. An example of a compound entity is a multiprocessor array, for which several different *templates* are available as library components. Currently available are one-dimensional array templates, several two-dimensional array templates with differing (pre-defined) indexing schemes, a three-dimensional torus and an Omega Network which can be instantiated for simulation at any hierarchical level. Indeed any entity can in principle be simulated at any specified hierarchical level. Figure 3 shows an example design window taken from an M.Sc. project [17] which has modelled the Stanford DASH architecture and its cache coherency protocols [18].

The DASH architecture consists of sets of processing nodes, grouped together in clusters of four and connected together via a common bus. Each node consists of an R3000 processor with a primary and secondary cache. As well as connecting the nodes together, the bus also provides access to memory which is shared between the processors and which forms part of the global address space of the system as a whole. Clusters are themselves connected together by a dual interconnection network. Figure 3 shows a four-cluster network in which the bold interconnection lines represent the request (?) and response (=) networks. The system is modelled as a three-level entity hierarchy. On the left of the figure two clusters are shown represented at the highest level, while the lower right hand cluster has been expanded to show the middle level (the dotted lines around a cluster of entities indicates expansion from a higher-level entity). The top right cluster has been further expanded to show the lowest level design of two of the nodes and also the lowest level design of the Directory Control logic, the subentities of which are responsible for ensuring inter-cluster cache coherence.

The HASE Architecture Description created in the design window describes the physical composition of the

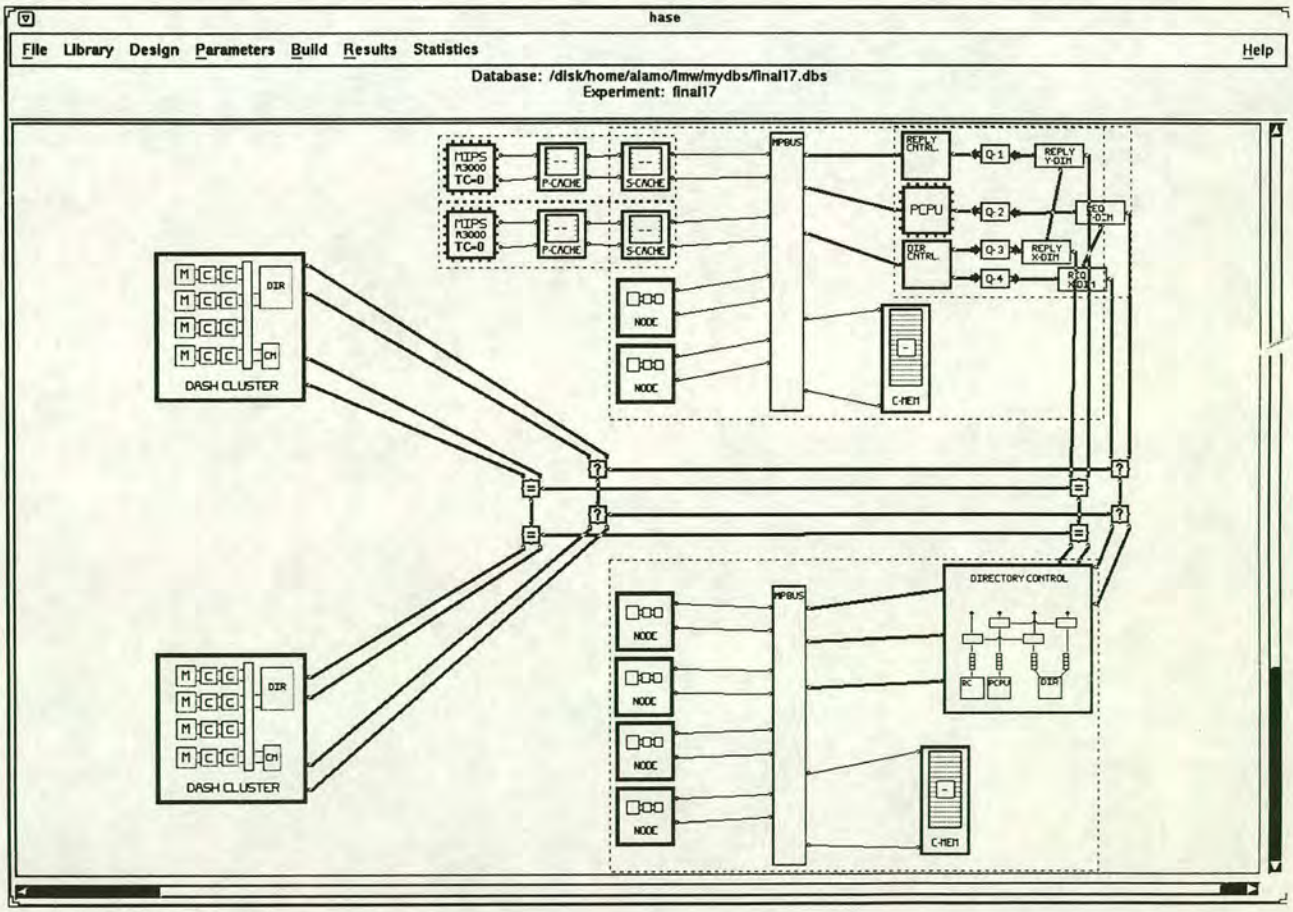


FIGURE 3. An example architecture in the HASE design window.

architecture. The behavioural aspects of the architecture are described by the corresponding Sim++ body methods. Sim++ is essentially C++ with functions to communicate events between entities. These events are sent to and received from ports which are the entity's link with the rest of the simulation. Entities may also have a variable number of parameters, which can be strings, integers, floating point numbers, enumerated types, structures, ranges, instructions or arrays.

To simulate a multi-processor system (say), the first task is to create an Architecture Description for it. The required entities (processors, memories, interconnection networks) are selected from a library, from a pre-defined HASE template or are custom designed. All entities can be further customized or modified from within HASE, including, but not limited to, additional subdivision (decomposition), grouping, and adding parameters. The required ports, including the link parameter are also added. The link parameter represents the data packet, message, instruction, etc., to be transmitted to/from the port over the link.

An entity may be defined at several different abstraction levels. The external interface (the set of ports) at each level must be identical, but at the lower level the entity may be composed of a set of interconnected 'sub-entities'. The abstraction level to be used for each entity is chosen at simulation time. Particular entities may be simulated at a lower level while leaving the rest at a higher level.

The entities are linked together to create the system to be simulated and each entity's behaviour is described in the Sim++ *body* method.

Global parameters can be defined for the system to be modelled. As the term implies, global parameters are accessible to all entities, for instance, the dimensions of a compound entity array.

The evaluation of an architecture normally involves the execution of test programs. An interface between simulations and a 'software level' is also needed for parallel programming or for investigating computational models. Several different approaches can be used within HASE; interpreting assembler, execution driven simulation, and interpreting a simple higher level language.

5.1. Instruction sets

One of the uses of HASE parameters is to store instruction sets. Instructions are typically divided into several different 'classes', such as load/store instructions, ALU operations, branches, etc. To deal with the resulting variety of instruction formats, HASE provides a special type of parameter, TInstrParam. For example, in

```
struct TInstr {
  TIClass iclass;
  union {
    char Name [20];
    Tmem_format mfield;
    Tbra_format ffield;
    Topr_format ofield;
    Tfopr_format ffield;
    int Word;
  };
};
```

the instruction class, iclass, is an enumerated type that indicates the type of the operands. The appropriate operand format (one of Name, mfield, bfield, ofield, ffield, Word) is used. The simulation code can then access the parsed instruction by checking the instruction class and referring to the elements of the relevant field. HASE can automatically produce a parser to load in data types which have been defined, e.g. to initialize a memory.

Higher level instruction formats can also be defined as HASE TInstrParam parameters. For example, a simple

```
if (stopping == 0) switch (Instr.OpCode)
{
  case COMPUTE:
      sim_hold( Instr.time, ev );
      break;
  case SEND:
      send_DATAPKT( TO_NET, ''TO_NET'', Instr.Pkt );
      sim_hold( SendTime, ev );     /* 'ev' = 'event' */
      break;
  case RECV:
      sim_wait( ev );
      SIM_GET(DataPacket,pkt,ev);
      sim_hold( RecvTime, ev );
      break;
  case STOP:
      stopping = 1;
      printf( "%s executed STOP instruction\n", sim_name() );
      break;
}
```

FIGURE 4. Sim++ switch statement.

```
$class_decls
    // Headers for extra functions
    int MPI_Send(void *,          /* data buffer */
            int,                  /* number of data elements */
            MPI_Datatype,         /* type of each data element */
            int,                  /* destination of message */
            int,                  /* message tag */
            MPI_Comm);            /* communicator */
    int MPI_Recv(...);
    // Any other function calls in the interface
$class_defs
    // Implementation of the extra functions
$body
    #include ''theactualcode.c''
```

FIGURE 5. Message passing interface code.

language might have **compute**, **send**, **receive** and **stop** constructs. These could be stored in memory and interpreted by a simple processor entity.

The simulation code can perform a *switch* statement on this field to determine the format of the commands. The Sim + + code segment in Figure 4 illustrates this last point.

Externally created code can be linked in with a HASE simulation. This enables the functionality of test programs on the simulated architecture to be used. Typically an interface is defined for the HASE object so the simulation can trap the desired function calls. Example applications include message passing interfaces and low level I/O on a simple computer system.

The basic form of the interface is as shown in Figure 5. In this example, the file *theactualcode.c* is standard MPI code making use of the functions. 'MPL_Send' and 'MPI_Recv'. In the simulation, these call the member functions which can be implemented in terms of the Sim + + primitives. In this way, standard code may be linked in with the simulation and can be used as a realistic workload.

Other ways of implementing this are possible, such as making the functions globally linked in rather than making them methods of the Sim + + object. It is even possible to link in Fortran 77 functions.

6. CODE GENERATION

Sim + + breaks down the simulation into an initialization and an execution phase. For inclusion in the code pertaining to both phases, HASE generates a Sim + + header file called ⟨*ProjectName*⟩*.h*. For the initialization phase, HASE generates the Sim + + initialization file *init.c*; for the execution phase, it generates the Sim + + entity constructors and bodies.

The behavioural specification for each entity at any given level of simulation is provided by the user in the ⟨*entityName*⟩*.sim* files. From these files and the Architecture Description HASE generates the Sim + + code required for the simulation. HASE also generates the makefile for compiling and linking the various component files.

- **The Sim + + body method**. Each entity in the model architecture needs a Sim + + *body* method for the specified level of simulation. If the entity is a compound entity, the default simulation level can be toggled. It is necessary for HASE to know at which level of decomposition the simulation will occur. HASE will then use the Architecture Description and the corresponding set of ⟨*entityName*⟩*.sim* files to generate the appropriate Sim + + code. The body can be constructed and edited off-line (external to HASE), or within the *Design Window* **Edit Body** function.
- **The project header file**. HASE generates a Sim + + header file for the Sim + + program (⟨*ProjectName*⟩*.h*) which contains parameter and event declarations, global constants, entity initialization structures, class definitions and declarations.
- **The initialization file**. HASE generates the Sim + + initialization file for the Sim + + program, called *init.c*, which contains the instances of the entities and allocates and initializes global data. The auxiliary functions for writing states to the trace file also reside in this file.
- **The Sim + + code**. HASE generates the ⟨*EntityName*⟩*.c* file for each type of entity in the Architecture Description based on information extracted from the entities themselves and the user defined ⟨*entityName*⟩*.sim* files.
- **The SMakefile**. The Sim + + makefile *SMakefile* used to compile and link the Sim + + code is also generated by HASE.

Dependency lists and compilation directives are constructed for *init.c*, the additional global functions file *global_ fns.c* (if it exists), and all ⟨*entityName*⟩*.c* files. The link directive to form the executable is also written to SMakefile.

7. SIMULATION

Running a simulation involves the execution of the Sim + + program produced by HASE in conjunction with the user specified ⟨*entityName*⟩*.sim* files.

Menu options under *Build* (Figure 3) within HASE trigger the generation of Sim + + code, compilation,

execution of the simulation and reading of the trace file.

A **debug** version of the simulation may also be compiled. This version of the code includes a simple routine which inserts commands to trace the current line number into the ⟨*entityName*⟩*.sim* file prior to compilation. Selecting this option pops up a window displaying the ⟨*entityName*⟩*.sim* file. This allows the program execution to be viewed alongside the animation.

8. DISPLAYING THE RESULTS

HASE provides two tools for viewing the results of a single simulation execution, an **Animator** and a **Timing Diagram**. Both assist in verifying the validity of the Architecture Description.

The **Animator** uses the event sequence held in an event trace file produced during a simulation run (normally the most recent) to provide the user with a visual display of activity in the system. It allows the data flowing between components to be visualized in a variety of ways, e.g. through moving icons showing individual instructions flowing down the stages of a pipeline or changes to the contents of a register bank when an instruction is executed. The important benefit of the animator is that it lets the user check that the model produces correct results. It is also useful as a presentation aid.

Animation is produced *automatically* from the simulation model with no need for the user to write explicit animation code. The simulation primitives for sending messages between components generate the trace information needed by the animator. It is also possible to animate a component's icon by providing different bitmaps for the different states. If a component has a

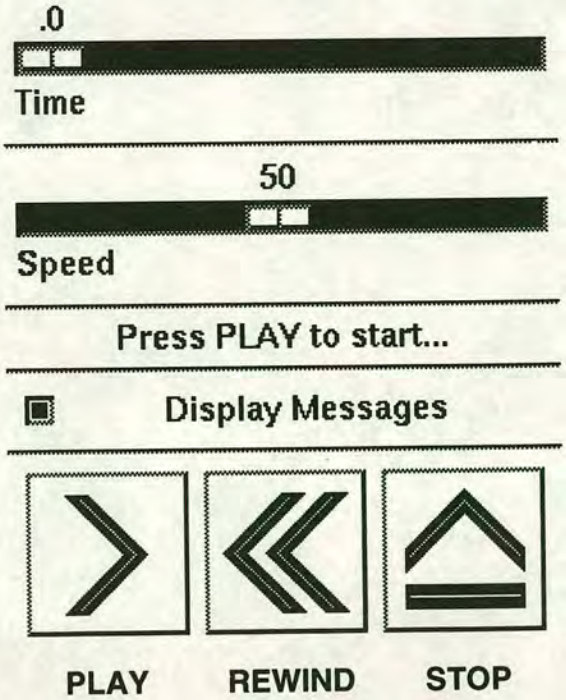


FIGURE 6. The animation controller.

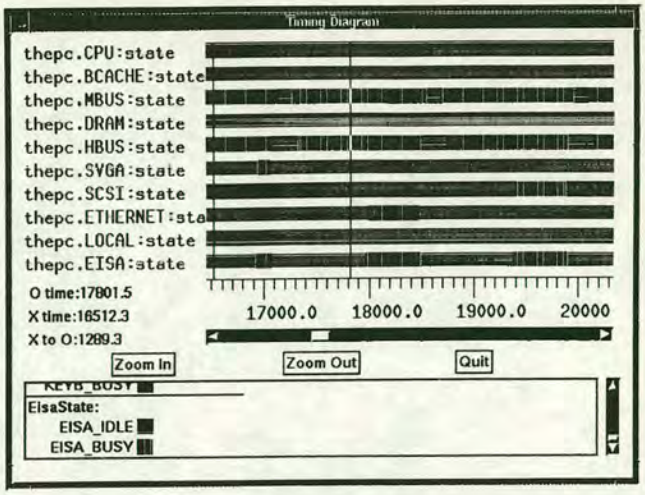


FIGURE 7. Example of a timing diagram.

state defined by the enumerated parameter **BUSY**, **ROUTING**, **IDLE**, animation is achieved by providing X bitmap files *BUSY.btm*, *ROUTING.btm*, *IDLE.btm*. Any number of a component's state variables may be displayed in this way. Variables may be 'dragged' onto the screen display using the component editor (or alternatively they may be left out of the animation altogether). Enumerated variables can be displayed either as the text value, or using bitmaps. The values of integer and string parameters are displayed as a text label. These values are updated whenever the user's simulation code calls the built-in function **dump_state( )**.

Manipulation of the animation of the architecture is handled through an Animation Controller (Figure 6) where time, speed and message display are handled as well as the standard 'tape' functions of PLAY, REWIND and STOP.

The **Timing Diagram** display (Figure 7) shows how the states of the *currently displayed* entities vary over the course of the simulation run. Only the enumerated parameters of each entity are regarded as the state. Different colours/patterns are allocated for each different state. Devotees of project management will recognize the display as a Gantt chart. Time measurements may be taken with two measuring bars, O and X and marked regions can be expanded to show finer detail.

There are additional single run *Data Collection Utilities* available through Sim + + that are not currently integrated with HASE, but still available to the user for manual inclusion in the simulation code.

9. METRICS

In general, the behaviour and validity of the project model are verified by single run results and the performance of the model is observed for subsequent tuning through experimentation with the model.

An experiment comprises repeated simulation runs varying input parameter values to produce output parameters from which performance statistics and other metrics may be gathered. General facilities are

provided for monitoring the values of particular state variables but more complex metrics may be obtained by explicitly writing Sim + + code.

Within the HASE environment the architect of the model defines the set of input parameters and also specifies a number of output parameters that could be monitored during the experiment. Users of the model can then determine which input parameters to assign values to in order to make certain observations regarding the performance of the model. The set of input or free parameters is a subset of the parameters of the model, chosen as being either external stimuli or interesting variable factors. The set of output parameters is the results obtained after applying the input parameters to the model. From the set of input parameters, single, sets of or a range of values can be specified.

The experimenter must decide what kind of statistical analysis should be performed on the partial results and view the final results to observe the performance of the model for the defined experimental state. HASE includes facilities for selection from a set of statistical functions and input of the confidence coefficient, interval width and maximum number of iterations. HASE also allows for and manages multiple experiments per model.

10. PROJECTS USING HASE

10.1. The ALAMO project

The ALAMO project (Algorithms, Architectures & Models of Computation: Simulation Experiments in Parallel Systems Design) aims to address the first of the Purdue Grand Challenges [1], 'to identify one "universal" or a small number of "fundamental" models of parallel computation that serve as a natural basis for programming languages and that facilitate high performance hardware implementations'. The project involves an investigation of the use of the H-PRAM model of computation [14] as a bridging model for parallel computation, i.e. an interaction platform for parallel software and hardware, via simulation. Algorithmic skeletons are written in an H-PRAM notation, compiled on to simulation models of parallel architectures created in HASE, and the performance metrics of various hardware architectures investigated. The goal is to determine the properties of cost-effective (cheapest possible) systems based on scalable architectures to provide efficient support of the H-PRAM model.

10.2. Parallel performance prediction

As an approach to satisfying the need for appropriate tools for developing concurrent applications for multiprocessors, HASE has been applied to parallel program development based on the MPI message passing interface. The ease of interfacing software layers to simulation models has made it straightforward to link actual code to models of an architecture. This approach to software development has also been investigated elsewhere using Proteus [20] and the WWT [21]. The advantages of using a simulation model for software development include repeatability, availability, variety, removal of Heisenbugs, ease of visualization and generality. At the design stage of parallel software it is better to have a simple method which is reasonably accurate than an accurate one which is unusable. Because of this, models have been calibrated using an MPI characterization routine which measures the performance of all MPI operations on an architecture. The focus has been on obtaining a quick first-cut design rather than on developing perfect models.

An interesting spin-off benefit of this project is that because Sim + + uses simple co-routines rather than Unix processes, the performance of a parallel MPI program running under HASE can be three to four times better than the same program running under a standard MPI distribution on the same workstation. The absolute improvement depends on the amount of context switching the program causes (since the context switch time for co-routines is faster than for processes).

10.3. An on-line teaching system for computer architecture

This project has produced a demonstration to aid students in the understanding of computer architecture. The demonstration involves playing back a pre-run simulation of the DLX architecture [22] which both animates the diagram of the architecture and displays a sequence of text windows which explain what is happening in the simulation. The simulation deals with hazards, multicycle operations, scoreboarding, etc. There is also a facility to enable students to create and animate their own programs.

11. CONCLUSIONS AND FUTURE DIRECTIONS

This paper has described a flexible environment for computer architects which has the following characteristics:

- **Hierarchy**: each part of a system can be both designed and simulated at the appropriate abstraction level.
- **Flexibility**: no system can anticipate all the needs of potential application areas; HASE has therefore been designed to be as flexible as the most common alternative—writing a simulation from scratch in a programming language.
- **Software support**: a simulation in HASE may involve both the hardware and software aspects of the systems under investigation i.e. HASE facilitates *software/hardware codesign* [23]; this is possible because software libraries can be linked into a simulation model.
- **Component reuse**: a major aim has been to make it easy to construct components which can be used in many different projects.
- **Graphics interface**: The X-Windows/Motif graphical

interface allows the user to view the results of simulation runs through animation of the design drawings.

HASE has already been used for a number of projects and users have commented on the relative ease with which they have been able to create their architectures. Further projects are in progress or are about to start. These include an extension of the on-line teaching system for computer architecture, simulation of a sparse vector processor and investigations of cache performance. Work on multiprocessor systems will include investigations of multiprefix algorithms and dynamic routing algorithms on mesh interconnection networks, and the evaluation of multiprocessor interconnection networks. In this project each of the different networks under investigation will be instantiated in a testbed consisting of a set of processor and/or memory components attached to the network. The processors will be relatively simple models, limited to generating network activity. The output from the various simulation runs will be used to visualize the effects of hotspots, for example, and to produce overall performance measures such as throughput and latency.

A number of possibilities for expanding the capabilities of HASE are also being considered. These include the incorporation of VHDL definitions and formal specifications as additional facets of HASE entities, and the incorporation of a flexible compiling system to allow experimentation with new instruction sets on meaningful example programs. In its simplest form such a compiler would offer modular flexibility in its back-end for generating code targetted at a pre-defined set of instruction sets. The ultimate in flexibility would be a compiler capable of compiling to an arbitrary instruction set, given the specification of that instruction set. Tools to support experimental compilation at some point on the spectrum between these two extremes will be investigated as part of related compiler research currently being undertaken at Edinburgh.

As well as providing a powerful tool for architecture research, HASE is also proving to be a valuable testbed for new ideas in modelling support environments. So far this work has concentrated on adding features to allow experiments involving replicated runs, and thus the exploration of parameter spaces, to be automated [24]. This is proving attractive in increasing the productivity of users, removing the need to repeat runs and collect results manually.

Further use of HASE is required before its run-time performance can be fully assessed, but it seems likely that improvements will be needed. One technique which is already being explored involves exploitation of concurrency mechanisms in the database to deliver results from multiple runs *in parallel* from a network of work-stations [25]. This should increase the speed of the system and allow more extensive simulations of more detailed models to be undertaken.

ACKNOWLEDGEMENTS

HASE is being developed as part of the ALAMO project supported by the UK EPSRC under grant GR/J43295. The authors would like to thank a number of colleagues, especially Paul Coe, Rob Pooley, Peter Thanisch, Nigel Topham and Lawrence Williams, who have provided constructive comments and input to this paper, and also all the students who have used HASE throughout its development for their comments and contributions. F.W.H. has been supported by an EPSRC Postgraduate Studentship.

REFERENCES

[1] Siegel, H. J., Abraham, S. *et al.* (1992) Report of the Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing. *J. Parallel and Distributed Comput.*, **16**, 199–211.

[2] Bell, C. G. and Newell, A. (1971) *Computer Structures: Readings and Examples*. McGraw-Hill, New York.

[3] McHenry, J. T. and Midkiff, S. F. (1994) VHDL modelling for the performance evaluation of multicomputer networks. In *Proc. MASCOTS-94*. IEEE Computer Society Press, New York.

[4] Buck J, Ha, S., Lee, E. A. and Messerschmitt, D. G. (1992) 'Ptolemy: a framework for simulating prototyping heterogeneous systems. *Int. J. Comp. Sim.*, 155–182.

[5] George, A. D. (1993) Simulating microprocessor-based parallel computers using processor libraries. *Simulation*, **60**, 129–134.

[6] Sheehan, K. and Esslinger, M. (1989) *The SES/sim Modelling Language*, Society for Computer Simulation, San Diego, CA.

[7] Evans, D. G. and Morris, D. (1992) Applying modelling to computer systems. In *Proc. IFIP 'CODES Workshop'*, Munich

[8] Nestor, J. A. (1993) Visual register-transfer description of VLSI microarchitectures. *IEEE Trans. VLSI*, **1**, 72–76.

[9] Zeigler, B. P. (1990) Object-Oriented Simulation with Hierarchical, Modular Models. Academic Press, San Diego CA.

[10] Robertson, A. R. and Ibbett, R. N. (1991) Simulation of the MC88000 Microprocessor System on a Transputer Network. In *Proc. EDMCC2*. Springer-Verlag, Berlin.

[11] Robertson, A. R. and Ibbett, R. N. (1994) HASE: a flexible high performance architecture simulator. In *Proc. HICSS-27*, Hawaii.

[12] Birtwistle, G. M. (1985) *Demos: Discrete Event Modelling On Simula*. Prentice-Hall, Englewood Cliffs, NJ.

[13] *Sim++ User Manual* (1992) Jade Simulations International Corp., Calgary, Canada.

[14] *ObjectStore Release 3.0 User Guide* (1993) Object Design Incorporated, Burlington, MA.

[15] Hillston, J. E. (1992) A tool to enhance model exploration. In *Proc. Sixth Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, Edinburgh.

[16] Pooley, R. J. (1991) The integrated modelling support environment a new generation of performance modelling tools. In *Computer Performance Evaluation Modelling Techniques and Tools*, Elsevier Science Publishers, Amsterdam.

[17] Williams, L. M. (1995) *Simulating DASH in HASE*, M.Sc. Dissertation, Department of Computer Science, University of Edinburgh.

[18] Lenoski, D. E., Laudon, J., Joe, T., *et al.* (1992) The

DASH prototype: implementation and performance. In *Proc. 19th Int. Symp. on Computer Architecture*, Queensland, Australia.

[19] Heywood, T. and Ranka, S. (1992) A practical hierarchical model of parallel computation I: the model. *J. Parallel and Distributed Comput.*, **16**, 212–232.

[20] Brewer, E. A. and Weihl, W. E. (1993) Developing parallel applications using high-performance simulation. In *Proc. IEEE Workshop on Parallel and Distributed Debugging*, San Diego, CA.

[21] Burger, D. C. and Wood, D. A. (1995) Accuracy vs. performance in parallel simulation of Interconnection Networks. In *Proc. ACM/IEEE Int. Parallel Processing Symp. (IPPS)*, Santa Barbara, CA.

[22] Hennessy, J. and Patterson, D. (1990) *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA.

[23] Razenblit, J. and Buchenreider, K. (eds) (1995) *Codesign: Computer-aided software/hardware engineering.* IEEE Press, New York

[24] Heywood, P.E., Pooley, R. J. and Thanisch, P. (1995) Object-oriented database technology for simulation experiments. In *Proc. Second United Kingdom Simulation Society Conf.*, North Berwick.

[25] Heywood, P.E., MacKechnie, G., Pooley, R. J. and Thanisch P. (1995) Object-oriented database technology applied to distributed simulation. In *Proc. EUROSIM Congr.*, Vienna..

[26] Lomow, G., Cleary, J. *et al.* (1988) A performance study of time warp. *Distributed Sim.*, **19**, 50–55.

CHAPTER 1

HIERARCHICAL ARCHITECTURE SIMULATION ENVIRONMENT

F.W. Howell and R.N. Ibbett

1.1. INTRODUCTION

The Hierarchical Architecture Simulation Environment (HASE) is a tool for modelling and simulating computer architectures. Using HASE, designers can create and explore architectural designs at different levels of abstraction through a graphical interface based on X-Windows/Motif and can view the results of the simulation through animation of the design drawings. This chapter describes the design and animation facilities of HASE, compares it with other simulation systems and concludes with suggestions for future tools based on several years' experience using HASE within the University of Edinburgh department of computer science.

1.1.1. The Motivation

Advanced simulation tools are available for low level electronic design, such as Spice for analogue circuits, and VLSI layout tools. However, tools for rapid prototyping of architectural ideas are less well established. Simulation languages can be used to model computer architectures, but the user has to be an expert on simulation. This is also the problem of general purpose simulation tools (e.g. SES/Workbench), where icons represent 'queues', 'servers' etc., and the link between a queueing model of an architecture and the architecture itself is not immediately apparent to the engineer not fluent in queueing theory.

Conventional languages (C, C++) are often used to construct simulators, but this approach involves starting from scratch for each new project. User interface aspects are often neglected as the tool will be thrown away with the next architecture. This is very wasteful, as many aspects of computers are constant between different architectures. The object oriented approach offers a solution. Standard components (such as memories, microprocessors and interconnection networks) can be held in a library. They can be constructed and linked together graphically on screen to create a simulation of an architecture, in much the same way that standard components can be wired together in a semi-custom VLSI tool. The difference is that the simulation is not fixed to low level wires and chip pins, but is free to choose the appropriate abstraction level.

HASE was designed to provide the flexibility of a raw programming language with the user interface advantages of a graphical tool.

1.2. DESIGN OF HASE

1.2.1. Overall operation

The HASE tool acts as a graphical front end to SIM++[1], 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 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 1.1 shows how the parts of the system fit together.

1.2.2. 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

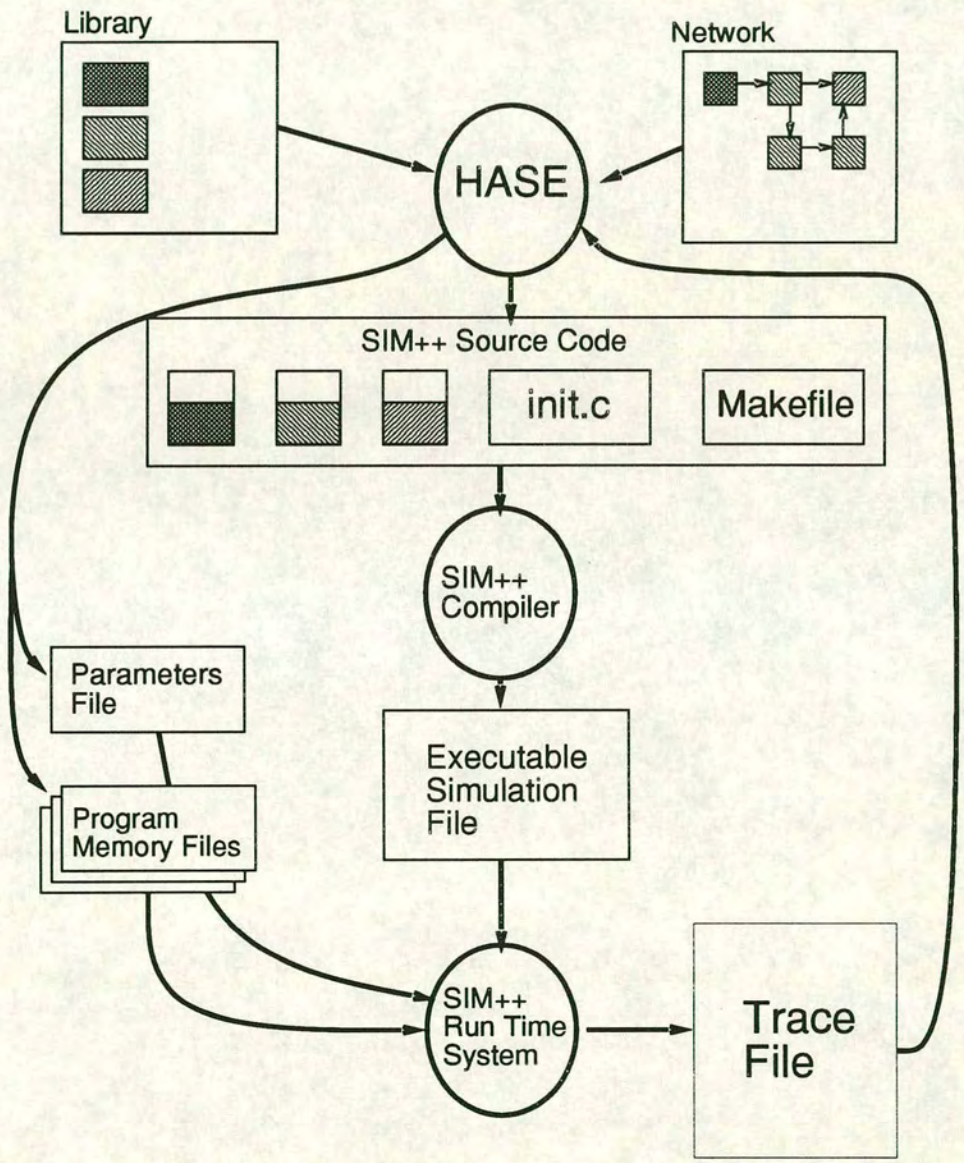


Figure 1.1. The top level design of HASE.

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 on SIM++ (which allows direct communication between any entities in the simulation) means that reusable components may be constructed with a defined interface.

- **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.

1.2.3. 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 1.2 illustrates two subdivided components connected by their external ports.

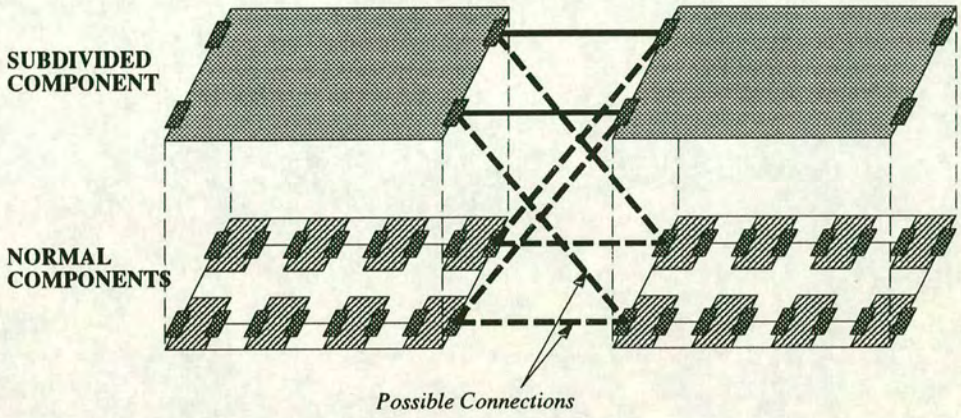


Figure 1.2. Two subdivided entities are connected by their external ports.

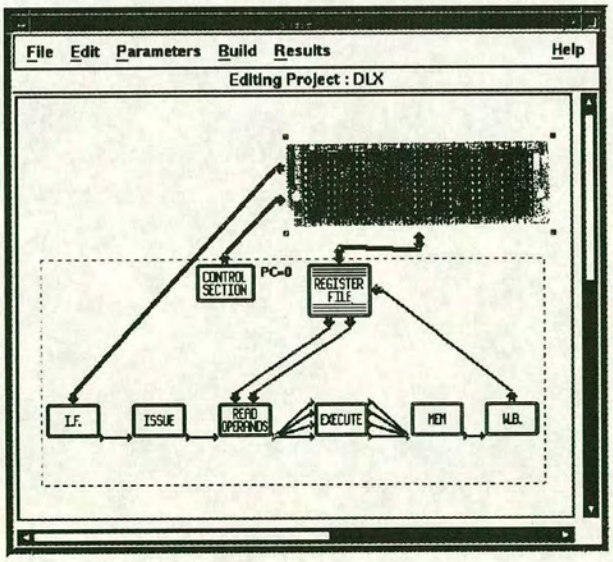


Figure 1.3. The HASE user interface.

1.2.4. 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 (see section 1.4.1. for more details). Array variables are 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.

1.2.5. 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.

1.2.6. 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 (see section 1.4.) 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.

Used in conjunction, these two tools 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 (section 1.4.1.). 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[2]. These runs are typically controlled by a Perl script and graphs produced using the GNUplot program.

1.2.7. Recycling Simulation Objects

One of the major benefits ascribed to object oriented techniques is that software components may be reused by others instead of being recreated from scratch.

This ideal has nearly been attained by hardware simulation systems; hardware components have well defined inputs and outputs so designs may be constructed by gluing together off-the-shelf components. The ideal is only "nearly" attained in this case as effort is still required to package components for others to use, so a certain amount of reinvention still occurs.

The situation isn't so rosy with object oriented software. This is partly because software is inherently more flexible than hardware. It becomes more difficult to define interfaces between objects when they aren't constrained to N physical wires, but may instead be composed of data types, interdependent methods, global variables and so on. It requires a significant investment in time and effort to document and prepare objects so others may use them[3]. As a result, few objects are generally shared between people, and most people only reuse code they have written themselves.

It was an early design aim of the Hase system to encourage object re-use as much as possible. This has met with some success in practice (but not as much as was hoped for). The interface to most Hase objects is by typed messages to ports, which makes reuse of objects simpler than the general C++ case (but not quite as straightforward as low level hardware models). Objects which play by these rules may be included in a project with no problems. However Hase does not enforce this model; it is possible for objects to use SIM++ techniques to communicate using global variables or to bypass the ports. This makes it more complicated to simply slot such an object into a project. Practicalities such as proper documentation being provided for objects also affect reuse.

The Hase library system has been designed to address these issues. Rather than storing a set of *components*, it stores a set of *projects* each of which includes a list of components, the parameter and message type

declarations and the global variables.

1.2.8. Object Oriented Databases

There has been substantial commercial and academic interest in object oriented databases recently. One common type of object oriented database is an extension to an object oriented language (such as C++) which provides for *persistence* of the objects. This approach is advertised as being suitable for storing the complex objects common in CAD systems, and providing desirable facilities such as version control and checkpointing of designs.

To investigate this approach to managing designs, Hase objects were made persistent by using the ObjectStore[4] database system. The experience was not without its problems. All HASE source files had to be preprocessed by the ObjectStore compiler before seeing the C++ compiler, which lengthened compile times. General run time performance became sluggish as all standard C++ pointers were replaced with persistent pointers, which could potentially result in a disk access. Any changes to class definitions made all previous database files unreadable (unless they were processed using a command line tool). Substantial source code modifications were required to be compatible with ObjectStore assumptions, and more modifications were later needed to obtain reasonable performance.

The conclusion from this experiment with object oriented databases is that the technology isn't yet mature enough for this type of CAD system. The general idea of allowing persistent objects within a language (without requiring I/O code) is a good one to be greeted with enthusiasm; in practice, however, adding an object oriented database requires much more effort than it would take just to write I/O code.

1.2.9. Limitations of graphical simulation systems

Die hard hackers sneer at graphical tools in general since they may never be as flexible as a programming language. This lack of flexibility is indeed a problem with *entirely* graphical tools which construct models at all levels by joining icons. At the lowest level of design, a description in a programming language is often best. However, there are also limitations with *entirely* textual descriptions; hardware and software designers usually use pictures to explain a system in terms of its subsystems. A compromise is therefore in order.

HASE is an inherently graphical system; if no pictures are needed, then there is little point in using it. However it does not impose a graphical

approach to the specification of individual objects. These are described in SIM++ and the full power of SIM++ is available to the programmer.

This compromise is finely balanced and it typically changes during the life cycle of a simulation project. Initially when the design is fluid, animation and graphics are very important for communicating ideas between researchers. Later, when the design solidifies, the important aspect is simulation run times for collecting experimental data.

1.3. OTHER APPROACHES

1.3.1. VHDL

VHDL has become established as the standard hardware simulation language. It enjoys support from all major EDA companies and provides for simulation at levels from behavioural down to gate level. This section compares the VHDL approach with using a C++ based simulation language for simulating hardware systems.

1.3.1.1. Why use anything other than VHDL? High level simulations incorporating software are usually written in C or C++ since these are the languages used by programmers. It is possible to link code from different languages, but the process is never entirely painless as interface routines have to be written to convert between the different data formats. The ideal is to use one language throughout. McHenry[6] uses VHDL for high level system modelling, and Swamy[7] describes object-oriented extensions to VHDL to make it more suited to system modelling.

VHDL incorporates very powerful features for modelling hardware; there are explicit constructs for wires `signals` and detailed timing information may be included. It's possible to detect glitches and other low level hardware problems.

At the software level, good support is also included for concurrent processes; e.g.

```
architecture behavioural of component is
  signal w : bit := '0';
begin
  proc1: process is
  begin
    w <= 1;
```

```
    wait for 10 ns;
    w <= 0;
    wait for 10 ns;
  end;
  proc2: process is
  begin
    wait until w = '0';
  end;
end behavioural;
```

Concurrent processes may be included *within* the description of a component. In SIM++, the unit of concurrency is the *entity* object. These entities communicate by sending and receiving *events*, which may contain data objects themselves. There is no concept of a *wire* as there is in VHDL, and no concept of a hierarchy of components (all entities are equal and may send messages to any other entity). The hierarchy is imposed on SIM++ by the Hase concept of ports. Programming in SIM++ is akin to programming a message passing parallel program.

The primary advantage of C++ based simulation languages (such as SIM++) over VHDL for system simulation is that linking to software libraries is significantly more straightforward. Basing communication upon messages passed between components rather than upon asserting signals allows a higher level view of the system, with the ability to send a data object at any abstraction level. VHDL on the other hand has much better tool support and standardisation than the various C++ simulation systems and includes direct support for modelling low level wire behaviour.

1.3.2. SIMULA / DEMOS

Another popular simulation approach is based on SIMULA and the discrete event package built on top of it (DEMOS). The original version of HASE was based on DEMOS[5]; the switch to SIM++ was motivated by the higher performance of C++ and the desire to interface to existing C and C++ libraries of code. Interaction between objects is based on shared *resources* which may have several operations defined, such as `wait`, `coopt` (a synchronisation).

1.3.3. Ptolemy

The Ptolemy project at Berkeley is a wide ranging simulation effort with a focus on signal processing[8]. It is a framework encompassing many different

simulation styles, including a discrete event domain. The package includes support for animations written manually using the Tcl/Tk toolkit.

1.3.4. Commercial Tools

Several commercial tools are available for network modelling and general system simulation, an example being BoNeS[9]. These tools present a slicker and more complete interface than research prototypes like HASE, but as their source code isn't freely available they are less suited to playing with new ideas and adding new features.

1.4. ANIMATION

Watch the cogs and pistons of a steam engine for a while and you get a feel for the workings of the machine. This is trickier with electronic systems; although they are many times more complex than the steam engine, they just appear to sit and work their magic without effort (bar the odd flashing light and smoldering component).

An animation of a simulation model can generate a similar intuitive feel for how an electronic machine works. This often suggests 'obvious' improvements and highlights design flaws which may be concealed by a flat diagram or descriptive paragraph. It is also fun to watch a complex design coming alive on screen and behaving as intended (or, as is more likely, *not* behaving exactly as intended).

The main reason that animation isn't usually an integral part of the design process is the amount of effort involved in building one. The problem with creating an animation separate from the main design is that changes to the design have to be made to the animation code as well. This makes the animation diverge from the actual design and become unusable.

Hase addresses this by making animation an *integral* part of design. Simple animations are generated automatically, based on the state changes of components and the messages which are passed between them. More complex animations may be customised to include `GIF` colour icons.

1.4.1. The Approach

Animation is based on the changes in value of a component's parameters. These may be dragged onto the screen using the component editor (figure 1.4); once this has been done, any time that parameter's value changes

it appears on the display.

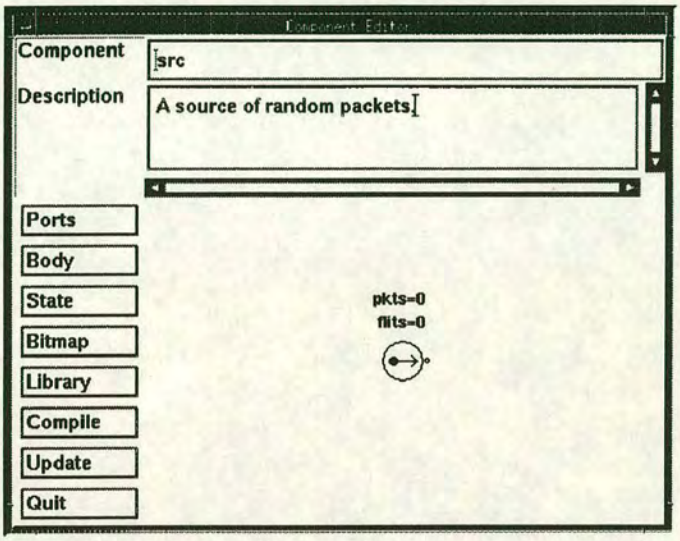


Figure 1.4. The component editor allows the state variables of an object to be dragged onto the display for animation.

The way a parameter is shown may be varied. `Value` just shows the value in screen (e.g. `123` for integers, `1.234` for floats, `BUSY` for enums). `Name+Value` shows the variable name as well (e.g. `curr_state = BUSY`).

Enumerated parameters may be displayed as icons instead of text; the icons are read in from bitmap files with the same name as the state (e.g. `BUSY.btm` or `IDLE.gif`). This is a simple but powerful technique for state animations; by simply providing the bitmaps for the corresponding states a customised animation is generated. These bitmaps may be displayed alongside the entity, or alternatively may be used to set the entity's bitmap.

`struct` parameters are displayed by drawing a box around the constituent elements (each of which may be displayed as above).

Thus far attention has been focussed on animating single parameters; any number of a component's parameters may be dragged onto the screen to be shown during animation, or they may be left hidden. It is also possible to define array parameters. The contents of these may be displayed on screen in a list box with a scroll bar and any updates or reads from the array are highlighted during the animation. Such updates are written to the trace using the `MEM_READ()` and `MEM_UPDATE()` macros in the SIM++ code. This technique has proved useful for displaying register contents and instruction buffers.

A simulation is not solely composed of state changes; there are also the messages sent between components. These messages may contain any form of data or handshake signal. The basic icon for a "message" may take any of the forms of static state parameters outlined above. This icon is animated by moving it down a link from one entity to another. The requisite line in the trace file is generated by the `send_DATA()` function in the SIM++ code, and the animation of the message is performed *at the time the message is sent.* Note that this is not necessarily the same as the time the message is acted upon by the receiving entity, as every SIM++ message is queued until the receiver is ready for it.

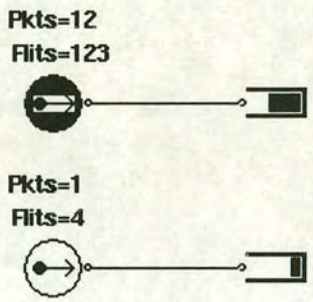


Figure 1.5. Changes in a component's state may be displayed on screen.

To show how the simulation code relates to the animation, figure 1.5 shows a `src` object connected to a `queue` and the following fragment shows part of the corresponding SIM++ code .

```
// excerpt from src.sim
  Pkts++;
  Flits++;
  if (ok_to_send)
    state=SRC_OK;
  else
    state=SRC_BLOCKED;
  dump_state();
  DataPkt d(123);
  sim_hold(1.234);
  send_DATAPKT(out,d,0.0);
  sim_wait(ev);
```

An example shows the format of the trace file which is generated on running the simulation and read in by the animator:-

```
// example trace file generated at run time
  u:src0   at 0.000: P SRC_BLOCKED 12 123
  u:queue0 at 0.000: P FULL_6
  u:src1   at 0.000: P SRC_OK 1 4
  u:queue1 at 0.000: P FULL_1
  u:src0   at 1.234: S out 123
```

Sometimes protocols require several messages to be exchanged between entities; in these cases it would be messy to animate all the acknowledge packets, so it is possible to send messages without generating any trace information. For large scale simulations, it is also often useful to avoid animating messages altogether and just show the state changes, so the "trace level" may be set to control which types of trace information are generated. The levels are:

comments and line numbers	1
message sends	2
memory updates	3
state changes	4
summary	5

Table 1.1. The levels of trace generation.

Setting the trace level to 4 (say) includes state updates and summary information in the trace, but not messages, memory updates or comments.

1.4.2. An example

Figure 1.6 shows an animation of a crossbar interconnection network with input and output queues. When the inputs block the icon is highlighted; it is possible to see the individual flits moving down the links and the queues grow and shrink dynamically.

1.5. APPLICATIONS

Architectural simulation work using the DEMOS prototype version of HASE is detailed in[5]. In 1992 work began on the current SIM++/Motif version which has been used in many MSc and final year honours projects, including simulation of multiprocessor WAN bridger/routers, simulation of the DLX processor and simulation of the DASH multiprocessor[10]. More details of

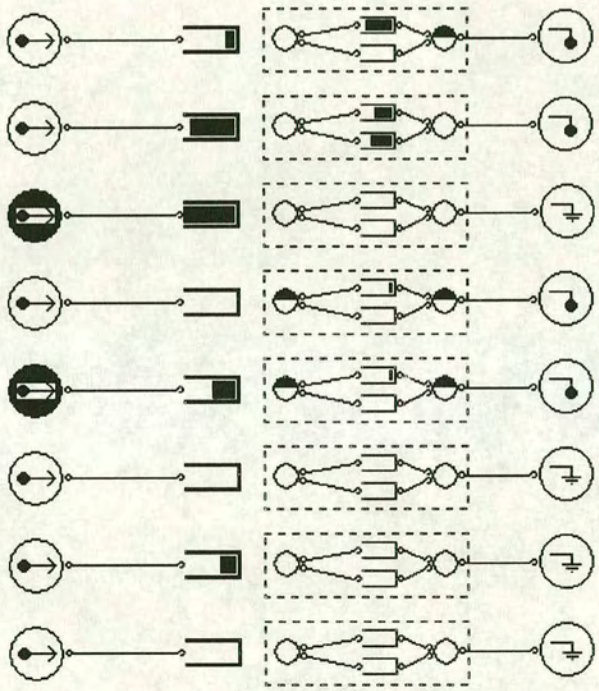


Figure 1.6. An interconnection network with input and output queues demonstrates the HASE animation facilities.

projects using Hase are given in[11]. Currently the main focus is on simulating multiprocessor interconnection networks and parallel MPI software. Many of the projects have involved linking simulation code to substantial existing libraries of C or C++ routines.

1.6. CONCLUSIONS

1.6.1. Important Messages

Animation has proved to be the most appealing feature of the Hase tool. The way in which it is incorporated into the design process allows swift construction of animation models and encourages communication and debate between designers. These advantages couldn't be obtained with an animation tool separated from the main design environment as there would be a problem maintaining consistency between the animation model and the one used for simulation.

The combination of an efficient threaded C++ with messages to communicate between objects is a powerful and intuitive programming model for software and hardware systems. It has also been useful that Hase imposes no restrictions on using SIM++ features.

The final message is that no simulation system will encompass all the needs of all projects. Many of the Hase features were included by students "extending" Hase to cope with the particular requirements of their project and this has proved the ultimate in flexibility, and a major advantage of having the source code and design available (which wouldn't be the case with commercial tools).

1.6.2. Future of the approach

New directions for the tool currently being investigated are closer tie-ins with an object oriented version of VHDL (to strengthen the links with hardware). VHDL itself is an attractive language for modelling hardware, but needs the addition of messages to model systems at a higher level. For software systems, it is very convenient to use a C/C++ like language since this makes it easy to include existing libraries of software.

Use of a parallel simulation language has been considered since the start of the Hase project and SIM++ originally had a timewarp version, but in projects to date the bottleneck hasn't been the simulation run time of individual runs, but rather the time to construct simulations. The lengthy simulations

have been successive runs with different parameters which have been run simultaneously on different workstations. We are currently experimenting with our own implementation of SIM++ to run on the Cray T3D to map out the performance of a model over a large area of the input parameter space in parallel.

REFERENCES

1. JADE INC, Sim++ User Manual, (Jade Simulations International Corp., Calgary, Canada, 1992).

2. J. HILLSTON, A Tool To Enhance Model Exploitation, Technical Report CSR-20-92, Dept. of Computer Science, University of Edinburgh, 1992.

3. B. STROUSTRUP, The C++ Programming Language (Addison-Wesley, 1991), 382-384.

4. OBJECT DESIGN INC, ObjectStore Release 3.0 User Guide, (Object Design Incorporated, Burlington, MA, 1993).

5. A.R. ROBERTSON and R.N. IBBETT, "HASE: A Flexible High Performance Architecture Simulator", in Proc HICSS-27 (IEEE, Hawaii, 1994).

6. J.T. McHENRY and S.F.MIDKIFF, "VHDL Modeling for the Performance Evaluation of Multicomputer Networks", in Proc MASCOTS-94, (IEEE Computer Society Press, New York, 1994).

7. S. SWAMY, A. MOLIN and B. COVNOT, "OO-VHDL: Object-Oriented Extensions to VHDL", IEEE Computer, **28:10**, 18-26 (1995).

8. J. BUCK, S. HA, E.A. LEE and D.G. MESSERSCHMITT, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", Int. J. Comp. Sim., **4**, 155-182 (1994).

9. S.J. SCHAFFER and W.W. LaRUE, "BONeS DESIGNER: A Graphical Environment for Discrete-Event Modelling and Simulation", in Proc MASCOTS-94, (IEEE Computer Society Press, New York, 1994).

10. L.M. WILLIAMS, Simulating DASH in HASE, (MSc Dissertation, Department of Computer Science, University of Edinburgh, 1995).

11. R.N. IBBETT, P.E. HEYWOOD and F.W. HOWELL, "HASE: A Flexible Toolset for Computer Architects", to appear in The Computer Journal, (1996).