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:-

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

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

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

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

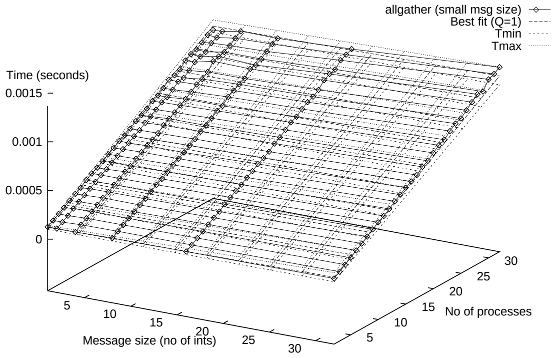
4 REVERSE PROFILING

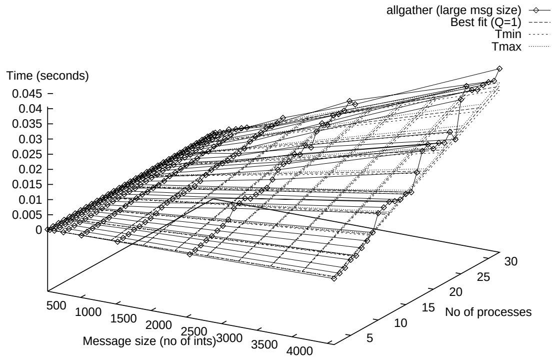
Reverse profiling is a technique which applies the MPI performance model for an architecture to a user's program to generate an estimate of the run time on that architecture. It uses the MPI profiling interface to intercept the user's calls to MPI functions and calculate the expected delay before returning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

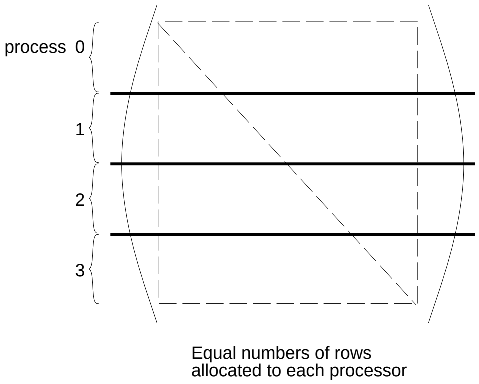


Figure 2 `outer` **: matrix distribution across 4 processors**

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.

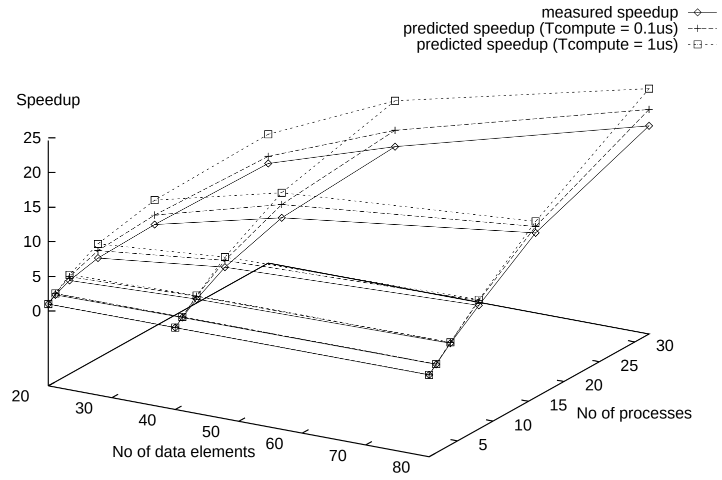


Figure 3 `outer` **: predicted and measured speedups on the Cray T3D**

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.