The Journal of Supercomputing, 34, 201–217, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

Communication Benchmarking and Performance Modelling of MPI Programs on Cluster Computers

D. A. GROVE duncan@cs.adelaide.edu.au
P. D. CODDINGTON paulc@cs.adelaide.edu.au
School of Computer Science, University of Adelaide, Adelaide, SA 5005, Australia

Abstract. This paper gives an overview of two related tools that we have developed to provide more accurate measurement and modelling of the performance of message-passing communication and application programs on distributed memory parallel computers. MPIBench uses a very precise, globally synchronised clock to measure the performance of MPI communication routines. It can generate probability distributions of communication times, not just the average values produced by other MPI benchmarks. This allows useful insights to be made into the MPI communication performance of parallel computers, and in particular how performance is affected by network contention. The Performance Evaluating Virtual Parallel Machine (PEVPM) provides a simple, fast and accurate technique for modelling and predicting the performance of message-passing parallel programs. It uses a virtual parallel machine to simulate the execution of the parallel program. The effects of network contention can be accurately modelled by sampling from the probability distributions generated by MPIBench. These tools are particularly useful on clusters with commodity Ethernet networks, where relatively high latencies, network congestion and TCP problems can significantly affect communication performance, which is difficult to model accurately using other tools. Experiments with example parallel programs demonstrate that PEVPM gives accurate performance predictions on commodity clusters. We also show that modelling communication performance using average times rather than sampling from probability distributions can give misleading results, particularly for programs running on a large number of processors.

Keywords: parallel computing, cluster computing, performance modelling

1. Introduction

Message-passing on cluster computers is the main programming paradigm used for high-performance scientific computing. The main reason for using parallel processing is to reduce the computation time required to execute what would otherwise be very long-running programs. Because poorly parallelised code reduces the performance that can achieved, there is great incentive to ensure that parallel programs are highly optimised. Unfortunately, a lack of sufficiently accurate and easy-to-use performance prediction methods for parallel programs has necessitated resort to a very time-consuming measure-modify design cycle to achieve this.

This scarcity of useful performance modelling methods is due to the notoriously complex behaviour of parallel programs, which makes it very difficult to devise adequate modelling methods. The main contributor to this complexity is contention, which causes non-deterministic delays and therefore non-deterministic program execution. It is very difficult to accurately predict the performance of parallel programs across a range of execution platforms with different numbers of processors and communication networks. Most existing performance modelling systems are either extremely slow or very approximate and do not accurately model the effects of network contention, which can significantly affect the performance of parallel programs running on large numbers of processors, particularly when using commodity Ethernet networks.

We have developed a new performance modelling and estimation tool called the Performance Evaluating Virtual Parallel Machine (PEVPM). PEVPM provides a simple, fast and accurate technique for modelling and predicting the performance of message-passing programs on distributed memory parallel computers. PEVPM focuses particularly on the accurate modelling of communication times, including effects of contention and potential non-determinism on program execution. To do this, it uses a virtual parallel machine to simulate the execution of a parallel program, based on simple annotations to the parallel program that could in principle be automatically generated. In order to accurately model the effects of network contention, PEVPM estimates times for low-level MPI communication routines based on the number and size of messages currently being passed through the network, which the virtual parallel machine keeps track of. Another important feature of PEVPM is that it samples from probability distributions of communication times, rather than using simple averages.

Existing MPI benchmark programs only provide average communication times, and have a number of other limitations. We have therefore developed an improved tool for benchmarking MPI communication routines, which can be used to provide detailed communications timing data to PEVPM. MPIBench uses a very precise, globally synchronised clock to measure the performance of MPI communication routines, which enables it to generate probability distributions of single communication times rather than averages of ping-pong measurements like those used by other MPI communication benchmarks. This allows useful insights into the communication performance of parallel computers, particularly the effects of network contention.

PEVPM and MPIBench are particularly suited to measuring and modelling the performance of parallel applications on clusters with large numbers of processors or of applications with significant communication overhead, where network contention and variability of communication times can be important. This is particularly important for clusters using commodity Ethernet networks, where relatively high latencies, network congestion and TCP problems (dropped packets and timeouts) can significantly affect communication performance. These effects are difficult to measure and model accurately using existing tools. Experiments with a variety of parallel programs with different communication patterns have demonstrated that PEVPM gives accurate performance predictions on a variety of cluster computers with different communication networks [9, 10].

In this paper, we give an overview of MPIBench and PEVPM, and present some results from the application of both of these tools to a standard scientific application running on a cluster with a commodity Fast Ethernet network. These results demonstrate that PEVPM can predict actual execution time to within a few percent (and usually within one percent). We also show that modelling communication performance using average times rather than sampling from probability distributions can give misleading results, particularly for large numbers of processors.

2. Measuring communication performance with MPIBench

There are a number of existing MPI benchmarking tools, including Mpptest [7], MP-Bench [22], SKaMPI [25] and the Pallas MPI Benchmarks [23]. These tools all determine the average communication time for point-to-point and collective communication operations using essentially the same approach: they measure the time taken for many repetitions of an MPI operation and then compute the average.

We have developed our own MPI communication benchmark, MPIBench [8, 9], which uses an accurate and globally synchronised clock to overcome some of the limitations of existing MPI benchmarks. Firstly, the globally synchronised clock enables it to to measure the communication performance characteristics of all of the processes in an MPI program, instead of simply measuring the round-trip time for point-to-point communications, or measuring completion times of collective operations at just a single process. Secondly, and crucial to the use of MPIBench results for performance modelling, the use of an extremely fine grained global clock allows timing data to be obtained for individual MPI operations, instead of just average times calculated over many repetitions of an operation. This gives MPIBench the unique ability to accurately quantify the performance variability of MPI operations due to contention, which it does by producing probability distributions (in the form of histograms) of the times taken to complete MPI communication routines. It is also possible to use parametrised functions to model the PDFs, based on fits to the histograms using standard functions [9].

3. Example results from MPIBench

MPIBench allows detailed measurement of all of the main types of point-to-point and collective communication operations in MPI. This section presents some MPIBench results for the `MPI_Isend` operation (and matching receive operation), which represents the performance of MPI's fundamental point-to-point communication mechanism. Detailed results from MPIBench for other MPI operations are presented in Grove's thesis [9].

The measurements described here were done on Perseus [11], a commodity cluster located at the University of Adelaide. Perseus has 116 dual processor nodes, each with 500 MHz Pentium III processors and 256 MB of RAM. Individual nodes are connected by commodity switched 100 Mbit/s Ethernet, built around five 24 port Intel 510T switches with stackable matrix cards that provide 2.1 Gbit/s of backplane bandwidth per switch. The software environment comprised of a RedHat Linux 6.2 base, glibc-2.1.3-15 and a Linux 2.2.12 SMP kernel and MPICH 1.2.0. MPIBench was run in a dedicated fashion, i.e. no other user programs or unnecessary system services were allowed to run during the benchmarking.

Figure 1 shows the average message-passing times for `MPI_Isend` operations over a range of small message sizes and for various numbers of communicating processes. Figure 2 shows the same information for larger message sizes. Lines of the same type (either solid or dashed) show results for an increasing number of nodes, while the different line types show the effects of using different numbers of processes on an SMP node. Lines are labelled $n \times p$, where n is the number of nodes and p the number of processes per node. The general effect of increasing the number of communicating processes per node or the total number

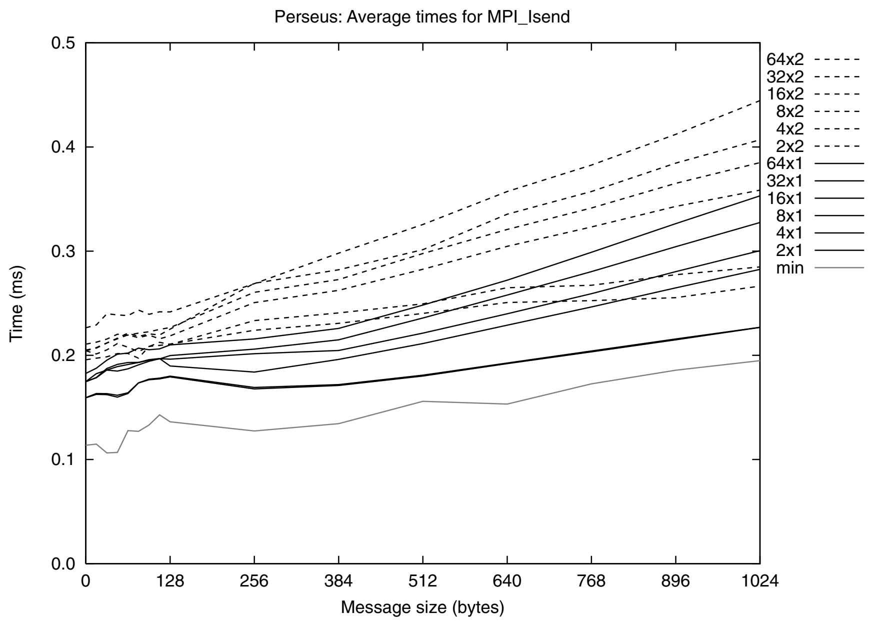

Figure 1. Average times for `MPI_Isend` using small message sizes with various numbers of communicating processes on Perseus.

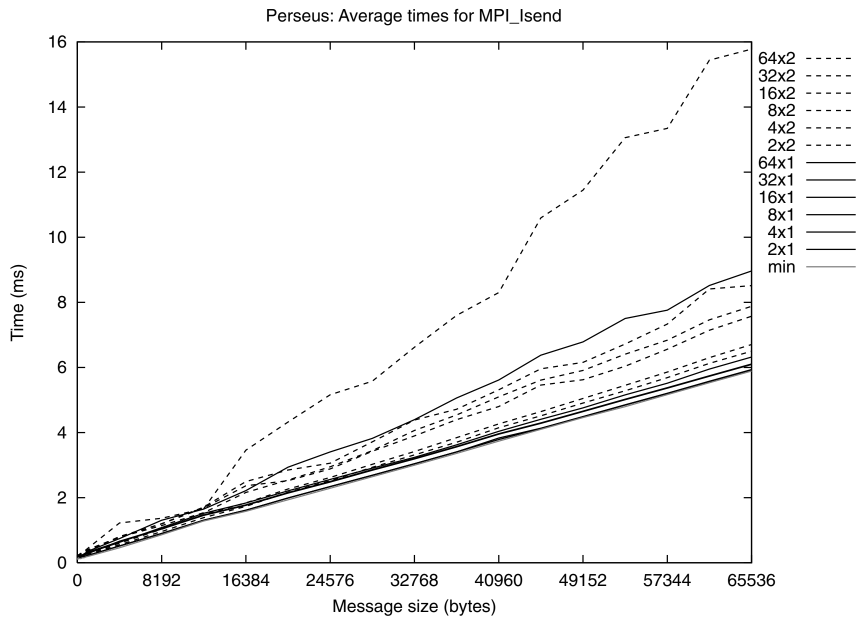

Figure 2. Average times for `MPI_Isend` using large message sizes with various numbers of communicating processes on Perseus.

of communicating nodes is to increase the level of contention. Lines of the same type (e.g. $n \times 1$ for different numbers of nodes n) can be easily distinguished by examining their ordering at the right hand side of the graph, which matches the ordering of the adjacent key. Increasing the number of processes per node increases the contention for the one network interface in each node as well as in the backplane network, while increasing the number of nodes only increases contention in the backplane network. Figures 1 and 2 also show another curve, labelled `min`, which indicates the minimum communication time that was observed between one pair of communicating processes. This minimum time represents the performance that can be achieved in the absence of contention.

The line marked 2×1 represents the performance of a simple ping-pong message, which is commonly used as a model of message-passing time. The similarity between minimum times and average times for this 2×1 case highlights the extremely small timing variations that occur when network congestion is eliminated. When this is the case, message-passing time T can indeed be closely modelled by the common approximation $T = l + b/W$ where l is the link latency in seconds, b is the size of the message in bytes and W is the effective bandwidth of the link in bytes per second. Closer inspection of Figure 2, however, reveals that there are actually two distinct segments to the data, with a knee occurring at 16 Kbytes. This is caused by the differences in the way that MPICH sends small and large messages.

Consider the effect on performance of contention in the backplane network, i.e. when there are a large number of communicating processes. For small messages, message-passing times become increasingly dispersed with increased numbers of communicating processes, which shows the susceptibility of the Fast Ethernet network in Perseus to contention. Figure 1 shows that, on average, transmission of a 1 Kbyte message takes 70% longer when 64×1 processes are communicating than when 2×1 processes are communicating. Modelling communication times by single-point values (e.g. from a ping-pong test) in situations such as this will lead to very inaccurate estimates. As the message size increases, however, the effect that the number of communicating processes has on delays in the backplane network becomes less noticeable (at least, until saturation of the network occurs) due to the longer time that the larger messages require for transmission.

Although a rough idea of the effects of contention on message-passing performance can be gauged from Figures 1 and 2, the effects are far more clearly demonstrated in Figure 3, which shows a number of performance distributions (plotted as probability distribution functions, or PDFs) that were recorded for 64×2 communicating processes exchanging messages between 0 and 1024 bytes in size. This shows how minimum and average communication times relate to the performance distributions that they approximate – the distributions have a relatively smooth rise from a bounded minimum time, through a peak which occurs very close to the average time and drop off fairly quickly to some maximum time.

Unlike the minimum time, which is bounded by the performance of a contention-free message, the maximum time is theoretically unbounded. Usually the tail of the PDF usually drops off so quickly that it can soon be treated as zero, as is the case for the `MPI_Isend` results presented here. Severe contention on an Ethernet network, however, sometimes leads to lost messages and thus retransmissions, which leads to outliers in the distribution at values related to the network's retransmission timeout parameters. MPIBench is able to account for these outliers because it measures the performance of individual messages, unlike other MPI

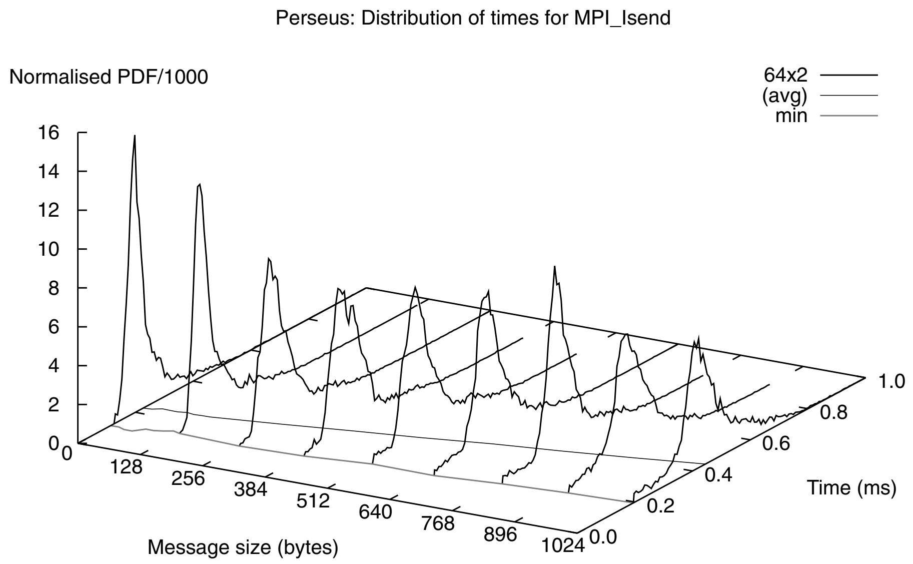

Figure 3. Sampled performance profiles for `MPI_Isend` using small message sizes with 64 × 2 processes (high contention for the local network interface and network backplane) on Perseus.

benchmarks. These outliers can have serious implications for program performance [9, 28] because the performance of most parallel programs is strongly influenced by their slowest process.

Severe performance degradation due to network saturation can be clearly seen in the long tails of the performance distributions in Figure 4. Figure 2 shows that this degradation starts to become significant for the 64×1 process case when message sizes reach about 16 Kbytes. For this 64×1 process case that is under consideration, three Intel 510T 24 port switches were spanned: two using 24 ports and one using 16. The onset of performance degradation began when a total of approximately 24×84.25 Mbit/s (since 81 Mbit/s is achieved between two processes for 16 Kbyte messages, plus 3.25 Mbit/s of Ethernet framing overhead) i.e. 2.02 Gbit/s was being delivered between the two fully utilised switches. Since the stackable matrix cards connecting these switches provide 2.1 Gbit/s of backplane bandwidth each, it seems that the backplane limit had been reached and the ensuing inter-switch saturation resulted in greatly reduced performance.

These results indicate that contention significantly affects the performance of small message transfers and that the effects of contention should only be ignored for large message transfers in the absence of network saturation. These performance characteristics must be taken into account during performance modelling of parallel programs in order to provide accurate performance estimates. The probability distributions generated by MPIBench can be used to accurately simulate message-passing communication times for the PEVPM performance modelling tool.

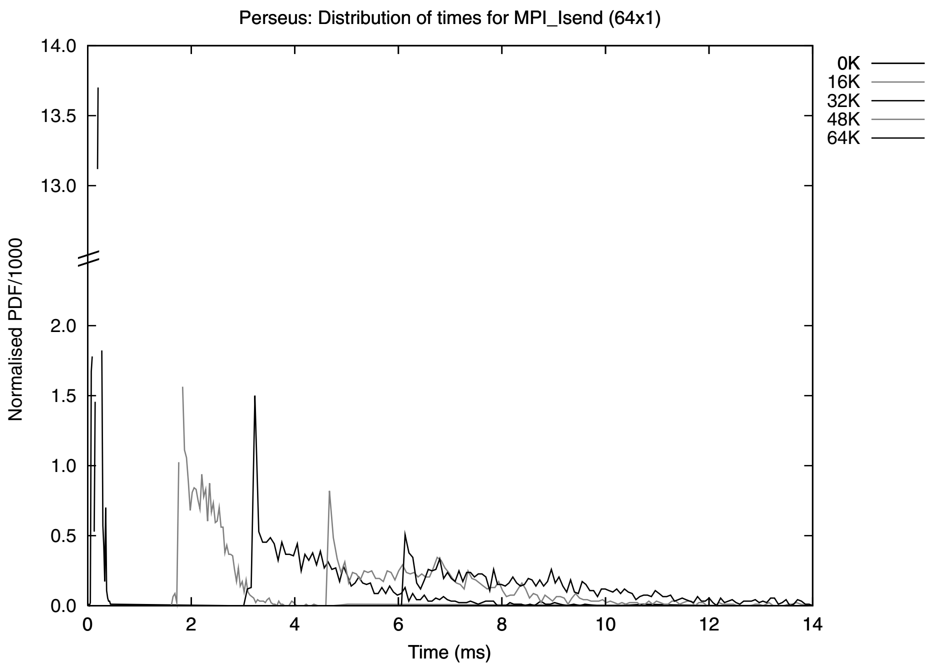

Figure 4. Sampled performance profiles for `MPI_Isend` using large message sizes with 64 × 1 communicating processes (high network contention) on Perseus.

4. Previous work on performance modelling

Performance modelling techniques can be classified according to a number of important characteristics: generality, expressive power, comprehensibility, accuracy and cost. While there are many existing performance modelling techniques that score well on various subsets of these characteristics, none do so for all of them.

For example, many research groups have developed enormously detailed models of specific parallel hardware/parallel software systems and used these to accurately predict performance. This includes efforts by Hoare [13], Milner [21] , Alur and Dill [2, 3], Fortune and Wylie [5], Kranzlmüller and Volkert [17], Schaubschläger [26], Magnusson et al. [19] and Hughes et al. [15] (see Grove's thesis [9] for a thorough review of these and other works). These sorts of performance models, however, suffer from a number of problems. They are typically very complex and time-consuming to create and are large and expensive to solve. They are not usually very flexible, so new models need to be constructed for every new situation. They can also be difficult to understand, because they are usually completely numerical rather than symbolic. Such models are therefore not very useful in the design stage of a parallel program, when it is necessary to quickly decide upon an effective parallel architecture and algorithm for a given problem from a very large number of possible choices.

At the other end of the spectrum, simple abstract models such as Amdahl's Law [4], Hockney's asymptotic r_∞ / $n_{1/2}$ model [14] and Grama et al.'s isoefficiency function [6] allow the performance of parallel programs under different conditions to be quickly and easily estimated. While these techniques provide reasonable ball-park estimates of performance in some cases, they are too simplistic to provide much useful information for most real parallel applications because they do not take into account any of the complex, non-linear effects such as contention and non-determinism which play such an important part in the performance of large parallel systems, especially on clusters.

It seems that the most promising modelling techniques lie somewhere in between. Even at this intermediate level though, all previous modelling techniques, such as those proposed by Adve [1], Mehra et al. [20], Parashar and Hariri [24], Jonkers [16], van Gemund [27], Labarta et al. [18] and Dunlop and Hey [12] fail to strike an ideal balance between accuracy, flexibility, implementation and evaluation costs (see Grove's thesis [9] for a detailed discussion of the advantages and disadvantages of each of these approaches). While the PEVPM modelling technique also adopts an intermediate level approach and draws upon the best ideas of many of the previous intermediate level techniques, it also introduces an intermediate level cost means of accounting for a number of performance effects that were previously only able to be modelled by detailed low level models. Therefore, in contrast to previous modelling techniques, the PEVPM modelling technique combines generality, ease-of-use, excellent accuracy and low evaluation cost.

5. Modelling MPI program performance with PEVPM

We developed the Performance Evaluating Virtual Parallel Machine to address the need for a sufficiently accurate and powerful yet generally accessible technique for estimating the performance of parallel programs. PEVPM focuses particularly on accurately modelling communication times in parallel programs, including the effects of contention and potential non-determinism in the program execution. PEVPM is based on a set of parallel program primitives, or building blocks, that can be used to compose the computation and communication structure of any message-passing parallel program. These primitives map to a set of performance directives that can be used to either annotate existing source code or to express some algorithmic idea in a standalone manner. A Performance Evaluating Virtual Parallel Machine executes a model of the parallel program based on these performance directives to simulate the time-structure of the program on some real or hypothetical parallel machine, thereby predicting program performance. Complete details of the PEVPM modelling system can be found elsewhere [9, 10].

Two properties make this essentially execution-driven simulation novel: its ability to abstractly simulate the direct performance effects of contention; and its ability to simulate the indirect performance effects caused by non-deterministic program execution due to that contention. This is achieved by dynamically creating submodels of individual computation and communication events using Monte Carlo sampling techniques, based on data dependencies, current contention levels in the system and detailed probability distributions of the performance of all low-level operations for a given parallel machine. These probability distributions can either be theoretical, or empirically determined by benchmarking low-level operations with MPIBench.

In order to determine the contention that will be faced by any particular message at any point in the execution of the program, PEVPM maintains a *contention scoreboard* that stores the state of all outstanding communication operations at any point in the simulation, including message sources and destinations, departure times and sizes. Message metadata is added to the contention scoreboard during PEVPM *sweep* phases, which simulate the execution of all running process until they each reach a *decision point*. Decision points occur when the execution structure of a particular process must diverge on the basis of dynamic program information, such as whether a particular message has arrived yet or not. When all processes have been simulated up to a decision point, a PEVPM *match* phase is initiated, which determines the arrival time of all messages in transit, based on the information stored in the contention scoreboard and the probability distributions of communication times associated with each of those messages. These probability distributions are a function of message size and the total number of messages on the scoreboard (i.e. contention level). Once the arrival times of messages are determined, these can be matched with an appropriate receive call, which will then determine what is to happen at the associated decision point and therefore allow program execution to continue during the next sweep phase. When matches occur, the messages they relate to are removed from the contention scoreboard. PEVPM evaluation operates as a series of interleaved sweep/match phases until no more decision points are encountered, which signifies the termination of the program being simulated. This modelling process enables the PEVPM methodology to produce highly accurate performance estimations for only a low-moderate evaluation cost.

Because PEVPM simulations evolve the program in virtual time, they automatically account for the effects of overlapping communication with computation, load imbalance and insufficient parallelism. Coupled with the ability to explicitly model communication and synchronisation losses and the associated resource contention issues of each (by sampling from distributions of communication times), the PEVPM methodology accounts for all the sources of both performance and performance loss in message-passing parallel programs. Furthermore, because all of these events can be annotated, PEVPM is capable of automatically determining and highlighting the location and extent of performance loss due to any source. In addition, it can also automatically discover program deadlock and help programmers trace down race conditions. Lastly, there is potential for the PEVPM methodology to be enhanced so that it produces entirely symbolic performance models rather than empirical ones, which would allow for even lower evaluation cost and would make the PEVPM approach even more attractive for very wide-ranging parametric-based performance studies.

6. PEVPM performance modelling of an example program

This section describes the PEVPM modelling of a parallelised Jacobi Iteration program on Perseus. Jacobi Iteration is a common parallel computing example because it is simple to explain yet has the same basic computation-communication pattern as all parallel algorithms with regular and local communication, including the large and important class of algorithms that perform stencil calculations on regular meshes of data. We have also tested the PEVPM using applications that are standard examples of the two other general classes of communication patterns in parallel programs: a Fast Fourier Transform as an example of a program with regular and global communication; and a bag of tasks (or task farm) as an example of a program with irregular communication. While the results for those applications are presented elsewhere [9, 10], the PEVPM provides similarly good performance predictions in those cases compared to the Jacobi Iteration example presented here.

Figure 5 shows the skeleton code for the Jacobi Iteration program, which includes all of the core computation and communication routines that have an effect on performance. Conceptually, every point in the `grid` is iteratively updated to be the average of its four neighbours (excepting boundary values, which do not change). Parallelism is introduced by a one-dimensional data decomposition that splits the `grid` into n subgrids, one for each of the processes involved in the computation. During each iteration, every process transfers the edge of its subgrid to any immediate neighbours in a regular-local communication phase and performs the computation on its subgrid in conjunction with any edge data that it received.

In a real problem, the iteration of communication and computation phases would terminate when some desired level of convergence was obtained between `grid` and `griddash`. Because that termination condition is data dependent, it could only be determined by actual execution of every computation, which would defeat the purpose of performance modelling. Therefore the example code simply terminates after 1000 iterations. This is perfectly reasonable, as performance comparisons between runs of the Jacobi Iteration on different configurations of a machine make far more sense on a per iteration basis. Note that since the PEVPM execution samples from PDFs of communication times, many iterations are needed to give an accurate average time per iteration. The PEVPM approach is like a Monte Carlo simulation of performance, and the number of iterations can be chosen so that the statistical error in the mean is negligibly small.

The skeleton code in Figure 5 has been annotated with PEVPM directives according the rules detailed in Grove's thesis [9]. This simple translation process took only a few minutes by hand, but could easily be carried out by an automated compiler with little or no human intervention. The PEVPM `Loop` directive specifies iteration; the `Runon` specifies code that should only run on certain processors, parameterised by *procnum* and *numprocs*; the `Message` directive stipulates a message transfer of a certain `type` and `size` between processes `to` and `from`; and the `Serial` directive defines the computation time required to execute a serial code segment.

There are a number of existing methods capable of accurately predicting the run-time of serial segments of computation. However, as the current version of PEVPM is mainly focused on accurately modelling the communication performance of parallel programs, an empirical model was used to abstract over the performance details of serial code segments. To simplify this process, a 256×256 `grid` size was chosen so that the problem size would always fit into cache memory when using 1 to 128 processors. This problem size also ensured that the times required for computation and communication were not so disproportionate as to render either one unimportant. One iteration of the stencil computation was actually run on Perseus and the execution time was measured. Because the per-processor amount of computation required for each iteration of the stencil calculations varies inversely with the number of processors available, the `Serial` computation time for each iteration of stencil computations was modelled by the measured execution time divided by *numprocs*. For larger problem sizes that do not fit into cache memory, the fraction of the problem size on each processor that fits into cache will vary with the number of processors used, causing a variation in the serial computation time. A simple approach for this type of regular

```
      int i, j, k, procnum, numprocs; int iterations = 100000;
      int xsize = 256; int ysize = 256/numprocs+2;
      float grid[size][size]; float griddash[size][size];
      MPI_Comm_rank(MPI_COMM_WORLD, &procnum);
      MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   // PEVPM Loop iterations = 100000
   // PEVPM {
      for (i = 0; i < iterations; i++){
   // PEVPM Runon c1 = procnum%2 == 0
   // PEVPM &     c2 = procnum%2 != 0
   // PEVPM {
        if (procnum%2 == 0){
   // PEVPM Runon c1 = procnum != 0
   // PEVPM {
          if (procnum != 0){
   // PEVPM Message type = MPI_Send
   // PEVPM &       size = xsize*sizeof(float)
   // PEVPM &       from = procnum
   // PEVPM &         to = procnum-1
            MPI_Send(grid[1], xsize, ... , procnum-1, ... );
          }
   // PEVPM }
   // PEVPM Message type = MPI_Send
   // PEVPM &       size = xsize*sizeof(float)
   // PEVPM &       from = procnum
   // PEVPM &         to = procnum+1
          MPI_Send(grid[ysize-2], xsize, ... , procnum+1, ... );
   // PEVPM Message type = MPI_Recv
   // PEVPM &       size = xsize*sizeof(float)
   // PEVPM &       from = procnum+1
   // PEVPM &         to = procnum
          MPI_Recv(grid[ysize-1], xsize, ... , procnum+1, ... );
   // PEVPM Runon c1 = procnum != 0
   // PEVPM {
          if (procnum != 0){
   // PEVPM Message type = MPI_Recv
   // PEVPM &       size = xsize*sizeof(float)
   // PEVPM &       from = procnum-1
   // PEVPM &         to = procnum
            MPI_Recv(grid[0], xsize, ... , procnum-1, ... );
          }
   // PEVPM }
        }
   // PEVPM }
```

(Part 1/2)

Figure 5. Skeleton code for the Jacobi Iteration with PEVPM annotations.

(*Continued on next page.*)

```
// PEVPM {
     else{
// PEVPM Runon c1 = procnum != numprocs-1
// PEVPM {
       if (procnum != (numprocs-1)){
// PEVPM Message type = MPI_Recv
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum+1
// PEVPM &         to = procnum
         MPI_Recv(grid[ysize-1], xsize, ... , procnum+1, ... );
       }
// PEVPM }
// PEVPM Message type = MPI_Recv
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum-1
// PEVPM &         to = procnum
       MPI_Recv(grid[0], xsize, ... , procnum-1, ... );
// PEVPM Message type = MPI_Send
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum
// PEVPM &         to = procnum-1
       MPI_Send(grid[1], xsize, ... , procnum-1, ... );
// PEVPM Runon c1 = procnum != numprocs-1
// PEVPM {
       if (procnum != (numprocs-1)){
// PEVPM Message type = MPI_Send
// PEVPM &       size = xsize*sizeof(float)
// PEVPM &       from = procnum
// PEVPM &         to = procnum+1
         MPI_Send(grid[ysize-2], xsize, ... , procnum+1, ... );
       }
// PEVPM }
     }
// PEVPM }
// PEVPM Serial on perseus time = 3.24/numprocs
     for(j = 1; j < ysize-1; j++){
       for(k = 1; k < xsize-1; k++){
         griddash[j][k]=0.25*
           (grid[j][k-1]+grid[j-1][k]+grid[j][k+1]+grid[j+1][k]);
       }
     }
     swap_ptr(grid, griddash);
   }
// PEVPM }
```

(Part 2/2)

Figure 5. (*Continued*).

application would be to estimate the computation times for serial code segments using the measured execution time of the sequential program for a problem size equal to the size of the sub-problem on each processor for the parallel program running on a specified number of processors.

We now evaluate the utility of the PEVPM methodology with regard to the model characteristics that were listed in Section 4. Firstly, the generality of the PEVPM methodology has been partially demonstrated through its applicability to a regular-local code. Demonstrations of the PEVPM's applicability to the other two possible types of parallel program, namely regular-global codes and irregular codes, can be found elsewhere [9, 10]. Secondly, the flexibility of the PEVPM approach has been implicitly demonstrated; because important program and machine parameters (such as *procnum*, *numprocs* and ostensibly data size arguments, by using appropriate compiler techniques) are retained symbolically in PEVPM models, those models can be easily re-evaluated under different input and environmental conditions. Thirdly, the demonstrated simplicity of adding PEVPM annotations to existing code, and its potential for automation, is testament to the low-cost of PEVPM model creation.

The PEVPM directives listed in Figure 5 were translated into a C language driver program, which therefore mimics the computation and communication structure of the actual Jacobi Iteration program. Because the driver program merely reflected the control structure defined by the PEVPM directives, it only took several minutes to generate by hand; note, however, that this process could be automated by using appropriate compiler techniques. This driver program was linked with an implementation of the PEVPM match/sweep algorithms and then run a number of times with different machine size parameters to predict the performance of the Jacobi Iteration for many configurations of Perseus. These performance predictions, presented as speedups, are plotted as dashed (or dotted) lines in Figure 6. The speedups observed while actually executing the Jacobi Iteration code on Perseus in corresponding situations were also measured and those results are plotted as solid lines in the same figure.

There are two classes of performance predictions in Figure 6. Firstly, there are PEVPM predictions made using performance distributions of constituent message-passing operations, which were measured with MPIBench as described in Section 3. These performance estimates (the dashed lines in Figure 6) are very accurate, always predicting completion time to within 5% and usually to within 1%. These predictions were consistently accurate, regardless of the number of processors used. This suggests that the small prediction errors that were observed were mainly due the granularity (i.e. histogram bin size) of the benchmark results that were input into the PEVPM simulation, rather than any underlying deficiency in the PEVPM approach. If desired, these errors could be reduced even further by using smaller bin sizes, at the cost of greater solution time requirements. Our constant and high accuracy predictions across such a wide range of machine sizes are unprecedented – other general performance modelling techniques rapidly lose the ability to make accurate predictions for parallel programs as the number of processors is increased [9].

One of the main reasons for inaccurate predictions by conventional modelling techniques when a large number of processors are involved can be seen in the second class of PEVPM predictions. These more simplistic performance predictions, shown by dotted lines in Figure 6, were made by inputting minimum or average benchmark results into the PEVPM evaluations instead of complete performance distributions. As denoted in the legend for the various results, these minimum and average times were garnered from MPIBench 2×1 process benchmarks, i.e. simple ping-pong benchmarks like those produced by existing MPI benchmarking tools, or by MPIBench $n \times p$ process benchmarks, which most existing MPI

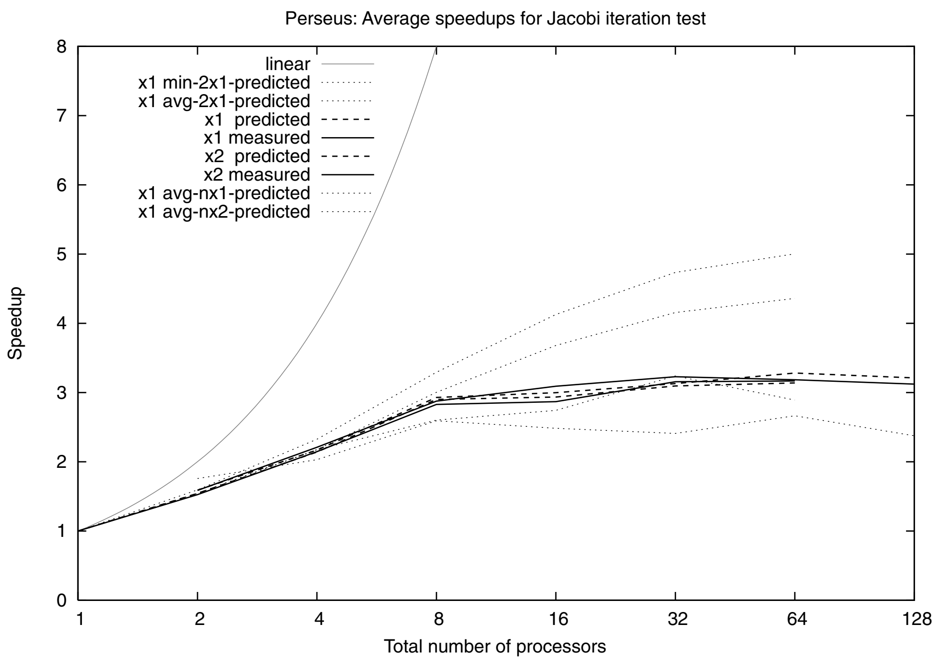

Figure 6. PEVPM-predicted average speedups and measured average speedups for the Jacobi Iteration test using $2 - 64 \times 1 - 2$ processes on Perseus.

benchmarking tools do not provide. These flawed predictions highlight the inaccuracies that result from using simple benchmark results.

Even for this simple regular-local program, the general performance prediction methods described in Section 4 are prone to large errors, which tend to grow in proportion with the total number of processors utilised. In particular, simplistic prediction methods utilising 2×1 process ping-pong data will always overestimate performance, because they do not account for the flow-on effects of contention in a parallel system. Using averages from MPIBench $n \times p$ process benchmarks to provide a crude accounting for contention will produce results of intermediate quality. How well this method will predict performance, however, depends on a number of ungeneralisable factors including how well actual communication performance can be approximated by a single value, the regularlity and granularity of the communication pattern employed by the application.

Hence, while simplistic performance prediction methods may possibly be useful for modelling the performance of parallel programs running on a small number of processors, they are inadequate for modelling large-scale parallel programs, where contention effects become very important. For parallel programs running on a large number of processors, accurate performance models must take the complete performance distributions of constituent message-passing operations into account, instead of just their ideal or average performances. This is especially important on clusters using commodity networks, where contention for network resources quickly limits performance.

With regard to model evaluation cost, the 11 hours and 15 minutes of processor time consumed by actually running the Jacobi Iteration program on Perseus were simulated in just under 10 minutes by our prototype (i.e. unoptimised) PEVPM implementation running on just one processor of Perseus. This comparison shows that PEVPM simulated the Jacobi program on Perseus at about 67.5 times its actual execution speed. Interestingly, the PEVPM algorithms themselves are close to embarrassingly parallel, so a parallel implementation of PEVPM would be even faster.

7. Conclusions

There are a number of limitations to the current tools that are being used for benchmarking the communication performance of cluster computers, and the tools that can be used for modelling the performance of parallel programs on those clusters. This paper presented an overview of two related tools that we have developed for more accurately measuring and modelling the performance of message-passing libraries and parallel programs on cluster computers.

MPIBench is a very useful tool for analysing the performance of MPI communication libraries on parallel computers. It can provide accurate and detailed probability distributions of communication times, rather than just averages. This is particularly useful for clusters using commodity Ethernet networks, where there may be significant variation in communication times due to network congestion.

The PEVPM approach to performance modelling provides a simple procedure for obtaining fast and accurate performance predictions of message-passing parallel programs on distributed memory machines. One of the key features of PEVPM is that it can simulate the execution of a parallel program by sampling from probability distributions of message-passing communication times generated by MPIBench, and thus take into account the effects of network contention. This is particularly useful when modelling clusters that use commodity networks, where communication overhead can have a significant impact on the performance and scalability of many parallel programs.

We have demonstrated that the PEVPM approach of utilising probability distributions of communication times provides very accurate estimates of the performance and scalability of MPI parallel programs, whereas modelling performance using average or minimum communication times can give misleading results.

The current implementation of PEVPM is still a prototype for demonstrate its general concept. Several of the steps in the modelling process must be done by hand. While these are all fairly straightforward, they could be quite time-consuming for large programs. We are aiming to automate most or all of these processes in future versions.

Acknowledgments

This research was supported in part by the Advanced Computational Systems and Research Data Networks Cooperative Research Centres, and a Faculty Small Research Grant. We are grateful to the South Australian Partnership for Advanced Computing (SAPAC) and the South Australian Computational Chemistry Facility for providing time on the Perseus cluster. Thanks also to Francis Vaughan for helpful discussions.

References

1. V. Adve. *Analyzing the Behavior and Performance of Parallel Programs*. PhD thesis, University of Wisconsin, Computer Sciences Department, December 1993.
2. R. Alur, C. Courcoubetis, and D. Dill. Model-checking for probabilistic real-time systems. In *Proceedings of the 18th International Conference on Automata, Languages and Programming (LNCS 510)*, 1991.
3. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–236, 1994.
4. G. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. *Proceedings of the American Federation of Information Processing Societies*, 30:483–485, 1967.
5. S. Fortune and J. Wylie. Parallelism in random access machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pp. 114–118, 1978.
6. A. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, 1993.
7. W. Gropp and E. Lusk. Reproducible measurements of MPI performance characteristics. In *Proceedings of the PVM/MPI Users' Group Meeting (LNCS 1697)*, pp. 11–18, 1999.
8. D. A. Grove and P. D. Coddington. Precise MPI performance measurement using MPIBench. In *Proceedings of HPC Asia*, September 2001.
9. D. A. Grove. *Performance Modelling of Message-Passing Parallel Programs*. PhD thesis, University of Adelaide, Department of Computer Science, January 2003.
10. D. A. Grove and P. D. Coddington. Modeling message-passing programs with a performance evaluating virtual parallel machine. *Performance Evaluation: An International Journal*, 60:165–187, 2005.
11. K. Hawick, D. Grove, P. Coddington, and M. Buntine. Commodity cluster computing for computational chemistry. *Internet Journal of Chemistry*, 3(4), 2000.
12. A. J. Hey, A. N. Dunlop, and E. Hernández. Realistic parallel performance estimation. *Parallel Computing*, 23(1/2):5–21, 1997.
13. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.
14. R. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, 17(10), 1991.
15. C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, February 2002.
16. H. Jonkers. *Performance Analysis of Parallel Systems: A Hybrid Approach*. PhD thesis, Delft University of Technology, Information Technology and Systems, October 1995.
17. D. Kranzlmüller and J. Volkert. NOPE: A nondeterministic program evaluator. In *Proceedings of the 4th International ACPC Conference (LNCS 1557)*, pp. 490–499, 1992.
18. J. Labarta, S. Girona, Pillet, C. A. T. V., and L. Gregoris. DiP: A parallel program development environment. In *Proceedings of the 2nd International Euro-Par Conference*, vol. II, pp. 665–674, August 1996.
19. P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Höllberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, February 2002.
20. P. Mehra, C. Schulback, and J. Yan. A comparison of two model-based performance prediction techniques for message-passing parallel programs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 181–189, May 1994.
21. R. Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science (92)*. Springer-Verlag, New York, 1980.
22. P. J. Mucci, K. London, and J. Thurman. The MPBench report. Technical Report UT-CS-98-394, University of Tenessee, Department of Computer Science, November 1998.
23. Pallas GmbH. Pallas MPI benchmarks home page. `http://www.pallas.com/e/produces/pmb/`.
24. M. Parashar. *Interpretive Performance Prediction for High Performance Parallel Computing*. PhD thesis, Syracuse University, Department of Electrical and Computer Engineering, July 1994.

25. R. Reussner, P. Sanders, and J. Larsson Träff. SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Computing*, 10, 2001.
26. C. Schaubschläger. Automatic testing of nondeterministic programs in message passing systems. Master's thesis, Johannes Kepler University Linz, Department for Computer Graphics and Parallel Processing, 2000.
27. A. van Gemund. *Performance Modeling of Parallel Systems*. PhD thesis, Delft University of Technology, Information Technology and Systems, April 1996.
28. F. Vaughan, D. Grove, and P. Coddington. Network performance issues in two high performance cluster computers. In *Proceedings of the Australasian Computer Science Conference*, February 2003.