Appeared in "Proceedings Supercomputing '93"

Analytical Performance Prediction on Multicomputers

Mark J. Clement and Michael J. Quinn
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

January 17, 1994

Abstract

Multicomputers have the potential to deliver Gigaflop performance on many scientific applications. Initial implementations of parallel programs on these machines, however, are often inefficient and require significant optimization before they can harness the potential power of the machine. Performance prediction tools can provide valuable information on which optimizations will result in increased performance. This paper describes an analytical performance prediction model. The model is designed to provide performance data to compilers, programmers and system architects to assist them in making choices which will lead to more efficient implementations. Efficient performance prediction tools can provide information which will help programmers make better use of the power of multicomputers.

1 Introduction

One of the most important advances in high performance computing is the increasing availability of commercial parallel computers. These machines promise to provide solutions to many problems that require more computational resources than are available on conventional sequential processors.

Because larger numbers of processing elements can be efficiently connected when memory is physically distributed, multicomputers have become increasingly popular in the scientific computing community. In the near future several vendors may produce multicomputers capable of delivering teraflop performance. With these massively parallel computers, scientists will be able to attempt to solve several of the "Grand Challenge" class problems that are currently limited by the speed of conventional computers.

The potential computational power of even the current generation of multicomputers is often not delivered on scientific problems that seem to be good candidates for parallel execution. It is often difficult for a programmer to predict what effect modifications to the algorithm will have on performance. Parallel system architects are forced to make trade-offs in system features without being able to predict the effect those decisions will have on the performance of important applications. This research addresses these two problems through an analytical model which uses application source code and essential machine parameters to satisfy the needs of these two groups.

High level parallel languages are essential in making parallel processors feasible for large programming projects. They allow the program to be written in a machine independent manner, and abstract away the complexity of explicit message passing. The Dataparallel C language [9] provides a SIMD model of parallel programming with explicit parallel extensions to the C language. Because of the static nature of Dataparallel C, it is possible to perform detailed performance analysis at compile time. The Intercom tool [13] developed for Dataparallel C identifies communication points in a parallel program. Using the information provided by Intercom and compiler generated information on the number of computations performed, this model can effectively predict the performance of Dataparallel C programs. The concepts developed here can also be extended to other parallel languages.

Several of the current generation of multicomputers have features which make performance prediction possible for parallel applications. This paper presents and evaluates a new analytical model and analyzes the success of the model in predicting the performance of parallel algorithms on different hardware platforms that share these features. This kind of performance prediction tool should enable programmers and compilers to take advantage of more of the potential power of multicomputers and make solution of "Grand Challenge" problems feasible.

2 Motivation

We have mentioned that performance prediction is important in achieving efficient execution of parallel programs. We will now explain how to use performance prediction information and how to derive the specifications for this model. We have found that taking an analytical approach as opposed to a simulation based approach to performance prediction improves the utility of the model. This model can be used by programmers to help in performance debugging, by compilers to choose the best possible optimizations, and by system architects to balance the communication and computational speeds. We will examine the needs of these three groups and show how their needs have influenced the design of this model.

- During program development, it is important for the programmer to determine the effect that changes to the source code will have on the performance of the algorithm. If this performance prediction is difficult or time consuming, the programmer will not be likely to try very many different implementations of an algorithm. Since the performance prediction of our analytic model does not depend on sample runs of the code, it can provide rapid feedback to the programmer. Several performance prediction tools require the programmer to rewrite the algorithm in a simulation language or to define the data dependency relationships. We feel that this is too high of a price to expect a programmer to pay. For this reason, this analytical tool extracts all of the algorithmic information from the source code of the program.

- Performing optimization of compiled programs on multicomputers can be more difficult than optimizing for shared memory systems. Data distribution, message passing costs, memory access overhead, and overhead induced by parallelization must all be considered to achieve an efficient implementation. An effective optimizer must have a model of the machine which takes all of these features into account in order to make correct decisions in compiling a parallel program. The analytical nature of our model makes it a prime candidate for use by an optimizing compiler. If simulation runs are required to make a prediction, the model will consume too much time to be useful in searching for the optimal combination of optimizations for each of these features. This model does not require simulation runs and has been shown to be effective in predicting the effects of optimizations on the performance of applications.

- Many parallel applications fail to achieve good performance on multicomputers because the systems are unbalanced. If the message passing time is too high compared to the time for a computation, then fine grain applications will never achieve good performance. System designers have little information about how balanced the communication and computation must be to perform acceptably on a target set of applications. Since this model does not require sample runs on the target architecture, system designers can use it to determine the effects of varying system parameters on a variety of parallel applications.

3 Performance Metrics

There are several ways of evaluating performance in a parallel environment. On a sequential machine, the principle performance goal is to minimize the execution time of an application. In a parallel environment it is also important to make efficient use of a large number of the available processors. In the best case situation, a parallel machine with p processors can reduce the execution time from the single processor time T_s to the parallel time $T_p = T_s/p$ for a speedup of p. If the parallel machine is not able to execute the algorithm significantly faster than a single node, then there is no reason to buy a parallel machine; it would be more cost effective for the algorithm to be run on a single processor. For this reason we will be using parallelizability, or relative speedup, as the performance measure which our model will predict.

Some research has suggested that scaled speedup is an important metric to use in evaluating parallel performance [8]. The scaled speedup metric measures the speedup attained when the problem size is scaled up along with number of processors. This metric is well suited to problems like weather prediction where the problem size can be easily varied. On problems where the problem size is fixed, or where the single processor execution time is unacceptable, relative speedup can be a better indicator [5]. Many scientific applications will require a drastic reduction in execution time before a computer solution will be practical. For these reasons, this research has concentrated on looking at the speedup attained on a fixed problem size when additional processors are added to the computation.

4 Developing the analytical model

An effective analytical model will incorporate features which have a dominant effect on performance and will ignore those features which have secondary effects. Several trends in modern distributed memory parallel systems permit us to make simplifying assumptions which lead to a more understandable model. We will use these assumptions to develop the parallel model used in this research. We have also identified several features in current multicomputers that need to be addressed if reliable performance prediction is to occur. These first order effects will be included in our model.

The time to execute the program on a single processor $T_{single} = T_s + T_p$ where T_s is the inherently sequential part of the algorithm and T_p is the parallelizable part of the algorithm. Several researchers [6] [16] have suggested the following form of a model for the parallel execution time of an algorithm:

$$\tau(p) = T_s + \frac{T_p}{p} + \sigma(p, topology)$$

where $\sigma(p, topology)$ is a function which estimates the communication overhead given the topology, and T_p is the time for the parallelizable part on one processor.

We have found that it is important to include a term for parallel overhead introduced by the emulation of virtual processors in Dataparallel C. Depending upon the choice of global or local variables, different optimizations are possible which result in variable overhead for virtual processor emulation. The number of times the compiler must set up a virtual processor emulation loop (N_o) can be used to estimate the parallelization overhead in the computation. The time spent in parallel overhead, T_o will be accounted for in our model.

The generalized form of the speedup for p processors can be expressed as:

$$S(p) = \frac{T_{single}}{\tau(p)} = \frac{T_s + T_p}{T_s + \frac{T_p}{p} + \sigma(p, topology) + T_o}$$

This reduces to Amdahl's law when $\sigma(p, topology) = 0$ and $T_o = 0$.

4.1 Trends in Floating Point performance

Traditionally, floating point arithmetic was so much more time consuming than integer arithmetic that the integer instructions were generally ignored in calculations of algorithmic complexity. The current generation of microprocessors exhibit floating performance that is equal to or greater than the integer performance. Many of the microprocessors used as compute nodes in multicomputers can execute two floating point operations (an add and multiply) in the same time that an integer instruction can execute. Several of the major multicomputer vendors are using this class of microprocessors for computational nodes. The Intel Paragon uses the i860 processor which has this feature [11]. The IBM POWERparallel machine uses RS/6000 technology which also has comparable times for floating point and integer instructions.

Since floating point and integer instructions take close to the same amount of time in these machines, it is possible for the model to estimate the number of computations through examining the parse tree generated by the compiler and counting the number of operations (N_{inst}).

4.2 Communication overhead

The $\sigma(p, topology)$ term incorporates overhead caused by communication between processors during the computation. In the general case, it accounts for effects caused by the topology dependent distance between processors, link bandwidth and message startup time for communications. For this analysis, we will assume that the machine uses cut-through or wormhole routing. With these circuit-switched routing schemes, the transfer time between any two nodes is fairly similar. Most modern parallel computers employ some form of circuit-switched technology to avoid the delay associated with store and forward routing. This simplifies the $\sigma(p, topology)$ term by allowing us to ignore distance considerations when estimating the communication cost for an operation.

One of the most significant contributors to communication overhead in the current generation of multicomputers is the message startup time ($T_{startup}$). We will define message startup time as the total time between when an application makes a call to the communication library and when data begins to be transmitted across the communication interface. This startup cost includes time spent in the communication library and system call overhead as well as the inherent time for the hardware to begin transmitting. As multicomputers have matured, they have added multitasking operating systems and more stringent error checking which have increased the overhead associated with starting a communication. Several researchers have noted that startup cost is the predominate factor in determining the total cost of communication [10] [17]. For this reason we will assume that overhead induced by limitation in actual bandwidth on the communication channels and link congestion are actually second order effects, and we will not consider them in our model. This makes the model much simpler, since it does not have to deduce the length of messages, but can just count the number of messages exchanged. Some applications which transmit large data sets will also see the the network bandwidth as a first order effect, but for many of the problems that we have dealt with it can safely be ignored. The Intercom tool can determine the number of communications $N_{communicate}$ from the source code. We will define the normalized startup cost $C_{startup}$ as the ratio $T_{startup}/T_{fp}$ (where T_{fp} is the time to execute a floating point instruction). The model will estimate the total number of cycles spent in communication to be $N_{communicate} * C_{startup}$.

4.3 Memory effects

Several researchers have noted that the memory hierarchy can have a significant impact on the performance achieved by a parallel program [7][18]. References to parallel (poly) variables in Dataparallel C are translated into structure accesses that will generally not be available in the on-chip cache. These uncached accesses will generally be limited to parallel code and have a significant effect on the performance of popular multicomputer processors including the iPSC/860 [15]. Using the number of uncached memory accesses extracted from the source code (N_m) and the number of cycles necessary to access an uncached memory location (C_m), the performance prediction tool can estimate the number of cycles spent waiting for uncached memory.

4.4 Compiler effects

Dataparallel C generates a standard C program as its output. The native C compiler then compiles the C code into an executable. The quality of the native C compiler can have a big effect on the number of machine instructions generated for each logical operation specified in the program. A constant for each compiler $C_{compile}$ can be determined through benchmarks or through extrapolating from results of the same compiler on other architectures. This compiler factor will be used to create a better estimate of the number of instructions executed in an application.

4.5 Applying the assumptions

Through applying the foregoing assumptions, we will develop an analytic model for predicting performance on multicomputers with wormhole routing, relatively high message startup costs and similar floating point and integer instruction times. Using information extracted from the source code, the compiler estimates N_s and N_p, the number of operations in the sequential and parallelizable portions of the code. Dataparallel C has explicit information about which parts of the program will be executed in parallel and which parts are sequential so this division is not a complex process for the compiler. Let $T_s = C_{compile} * N_s * T_{fp}$ and $T_p = (C_{compile} * N_p + C_m * N_m) * T_{fp}$.

Our model of $\sigma(p, topology)$ involves only the startup cost $T_{startup}$ and the topology. We can express $\sigma'(p, topology) = \sigma(p, topology)/T_{fp}$ in terms of the normalized startup time. For a broadcast communication on a hypercube topology $\sigma'(p, topology) = N_{communicate} * C_{startup} * (1 + \log(p))$.

Using the dominant effects we have described here,

$$\begin{aligned} T_s &= (C_{compile} * (N_s + N_p) + C_m * N_m) * T_{fp} \\ T_p &= C_{compile} * N_s * T_{fp} + \\ &\quad \frac{(C_{compile} * N_p + C_m * N_m) * T_{fp}}{p} + \\ &\quad T_{fp} * \sigma'(p, topology) + T_{fp} * C_{compile} * N_o \end{aligned}$$

With $S(p) = T_s/T_p$ the T_{fp} terms drop out and we are left with a speedup equation dependent only on the variables which are available to our prediction tool.

The terms N_s, N_p, N_m, N_o and $N_{communicate}$ can all be determined from the internal parse tree generated by the compiler. The term $C_{compile}$ can be determined through a sample program or through experience with the compiler on other processors. $C_{compile}$ describes the efficiency of the compiler in generating optimized code. The term $C_{startup}$ can be determined from a sample communication program or estimated from machine specifications. C_m is generally available as a system specification.

5 Experimental Results

Several experimental results are presented here to validate the concept of a analytical performance prediction model. One set of experiments was performed to determine if the tool could accurately predict the performance effects of changing the implementation of an algorithm on a fixed target machine. This kind of performance prediction information would be used by a programmer or compiler to optimize a program. Other experiments were performed to demonstrate that the tool could predict performance on different target machines for several different algorithms. This kind of prediction information would be useful to system designers in determining the effects of changing system parameters.

5.1 Source code variation

One of the challenges of programming in Dataparallel C is determining the parallel type to use for different variables. Dataparallel C has a notion of global (mono) variables which are kept consistent across all of the physical processors and local (poly) variables which may be different for every virtual processor. There is a complex set of rules for determining which parallel type to use for loop variables or array index variables to produce the best performance [9]. In some cases, the choice depends on the target architecture to be used by the application. If the compiler were able to predict the performance characteristics for each of the choices, it could automatically select the correct types and relieve the programmer of the task of variable type selection.

Matrix multiplication is often used as a benchmark on parallel machines. Several versions of the matrix multiplication algorithm have been implemented in Dataparallel C. As a test of the model we changed two of the loop indices in the "matrix2" implementation from parallel local variables to parallel global variables. The experiment was performed on the Intel iWarp array. The iWarp is connected in a mesh topology, uses wormhole routing, has a message startup latency of 470 cycles and has similar floating point and integer execution times. The prediction tool was able to accurately predict the performance of the original version and the new version called "matrix2+" . The results are shown in Figure 1.

5.2 Experimental results on different target machines

A second set of experiments were performed using two target architectures that exhibit the features we described in our model development. The experiment was performed on the Intel iWarp array and on an iPSC/860. The iPSC/860 uses the Intel i860 processor, is connected in a hypercube topology, uses wormhole routing, has a message startup latency of 5280 cycles and has similar floating point and integer execution times. Experiments were performed using three standard Dataparallel C applications. The experimental results show that the analytical model is successful in predicting performance for the two machines.

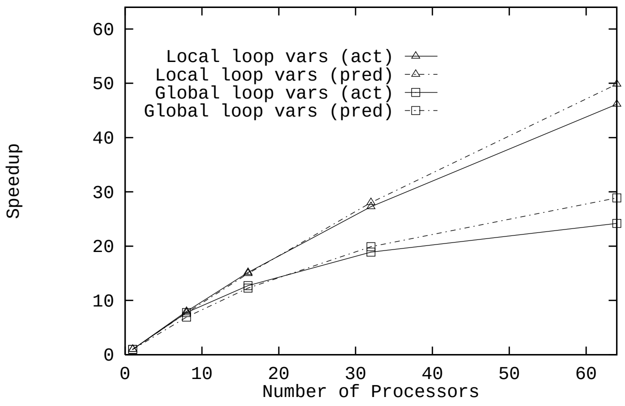

Figure 1: Experimental and predicted values results for 256x256 matrix multiplication on the iWarp array.

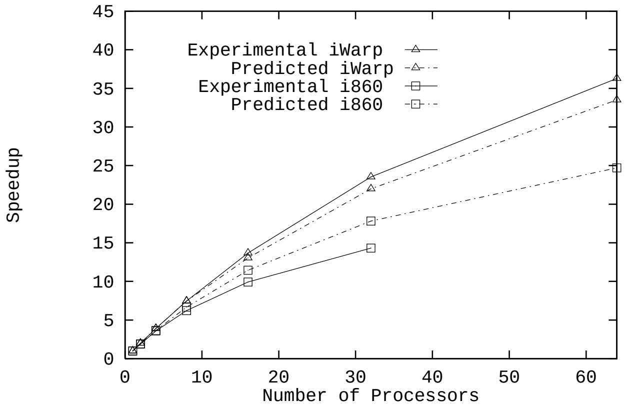

Figure 2: Experimental and predicted results for the shallow water atmospheric model.

5.2.1 Atmospheric model

This application was developed by the National Center for Atmospheric Research for benchmarking the performance of parallel processors [9]. The program solves a system of shallow water equations on a rectangular grid using a finite differences method. The model uses a two dimensional array of data elements that communicate with their nearest neighbors. The performance prediction tool is able to approximate the actual performance fairly accurately. More significantly, the tool was able to clearly differentiate between the performance to be expected on the two machines. The results are shown in Figure 2.

Performance information from an analytical model can allow a system architect to observe the effects of

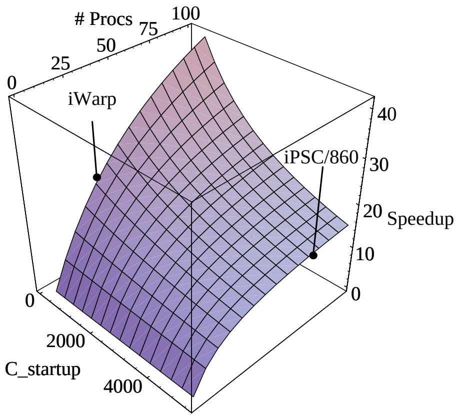

Figure 3: Speedup results for the shallow water atmospheric model with variable message startup cost. C_startup is the message startup time divided by the time for an arithmetic operation. A C_startup value of 470 corresponds to the iWarp processor and a value of 5000 approximates the iPSC/860.

changing specific system parameters. In Figure 3 the message startup cost is varied for the shallow water atmospheric model to show the effect this parameter has on speedup. Figure 4 shows how cache miss penalty and message startup cost interact in predicted speedup results. Machines with large cache miss penalties will achieve larger speedup values for a given message startup cost. The performance in Mflops will be seriously degraded on machines with large values of C_m as is shown in Figure 5. Performance results from a large number of parallel applications should make the trade-offs much clearer to systems architects.

5.2.2 Ocean Circulation model

This program simulates ocean circulation using a linearized, two-layer channel model [9]. This application also uses nearest neighbor communication but in this case the two machines achieve nearly identical speedup results. This is due to a combination of grain size differences and differences in the number of accesses to uncached memory between the ocean circulation model and the shallow water model. It would be difficult for a programmer to guess that the two programs would perform this differently from perusing the source code. Again, the performance prediction tool was able estimate the speedup attained by the application. The results are shown in Figure 6.

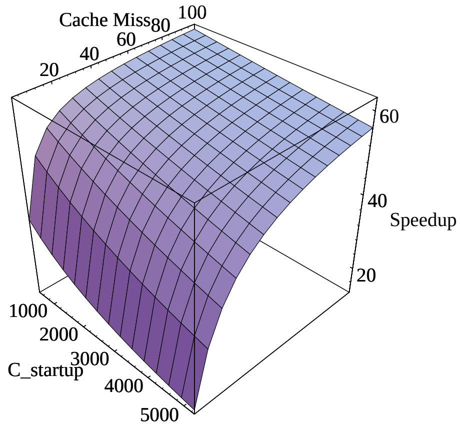

Figure 4: Speedup results for the shallow water atmospheric model for 64 processors with variable message startup cost and cache miss penalty.

5.2.3 Sharks World

Sharks world is included as an example of an application with few communications. The program simulates sharks and fish on a toroidal world [9]. As expected, both machines are able to achieve near linear speedup on this application. The predicted and actual results are shown in Figure 7.

6 Related Research

Several different approaches have been taken in modeling parallel systems and predicting performance. Most of the analytical models do not use application source code and so are limited in their accuracy. The simulation based prediction tools do not allow system architects to experiment with different trade-offs in system parameters.

Markov models have been used to approach the problem from a queueing theory direction. Kapelnikov describes a methodology used to build Markov processes, starting from the description of a program [12]. Building Markov processes requires more time and expertise than most programmers have available, especially for large programs.

Balasundaram et al. [2] have developed a performance estimator based on a training set approach. Their analysis focuses on determining the best data distribution for a given algorithm. System parameters are determined using training sets which are similar to the sample programs which could be used with this research to determine the startup cost. The training set

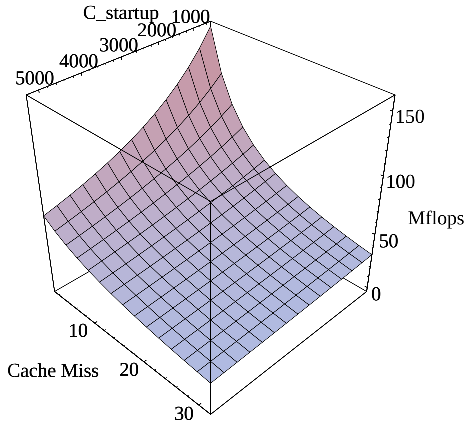

Figure 5: Mflops results for the shallow water atmospheric model for 64 processors and variable message startup cost and cache miss penalty. A processor speed of 10Mflops is assumed for each processing element.

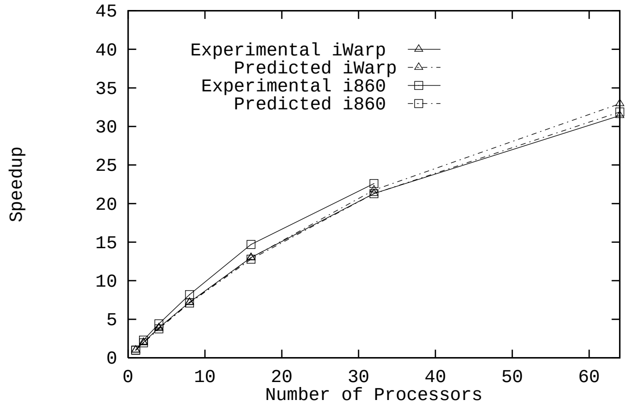

Figure 6: Experimental and predicted results for the ocean circulation model.

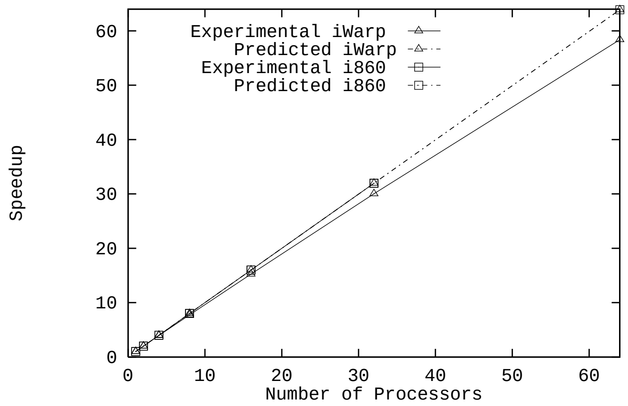

Figure 7: Experimental and predicted results for Sharks World simulation.

approach combines several of the effects that must be separated to allow for architectural experimentation. They also do not account for the cache miss penalty, which is important in the class of processors which we are studying here.

Morris has developed a data flow modeling language, which is used to model a computation [14]. The requirement that a programmer must rewrite the algorithm in a new language will not only limit the ease of use of this performance tool, but will limit the accuracy, since the real program will not be examined.

Born has developed an analytic model that relies on some statistical distribution of communication requests [3]. Traditional analytic models that do not use information from program source code give little insight into the performance of a machine on an actual application.

Annaratone has developed a tool that uses the communication/computation ratio to make decisions in parallelizing Fortran code [1]. The tool uses an initial run of the program to determine system and algorithmic parameters. This tool provides a good example of how performance prediction information can be used to optimize compiler translation of source code.

A static parameter based prediction tool was developed by Fahringer [4]. This tool uses a sample run on the target machine to determine model parameters for the system and algorithm. These parameters are then used to optimize the parallel implementation of Fortran 77 programs.

One principle difference between our model and other research is that it allows for performance prediction without a training run on the target machine. It is important for system architects to be able to predict performance on future machines and other tools do not provide this functionality. Parallel computer purchasers can also benefit from being able to estimate the performance of their applications on machines with different system parameters.

7 Conclusions and Future Work

Predicted performance information can be useful in optimizing multicomputer system performance. In this paper we have examined several features of current multicomputers which make it possible to predict performance for these machines. An analytical multicomputer model was developed, which allows us to gather performance information with minimal programmer intervention. This model is particularly useful, since it does not require sample runs on an existing machine and can be used to predict performance on a hypothetical machine or one to which the user does not have access. Performance data from the model corresponds closely to actual data acquired from two commercial multicomputers.

Future work will concentrate on incorporating this simplified model into the existing Intercom tool and testing the tool on a wider range of applications. An attempt will be made to classify applications whose performance is accurately predicted by this analytical model. Additional analysis will also be performed to determine new model features which will allow performance prediction on a broader range of parallel programs and hardware architectures.

Much work is left to be done as far as generalizing the model to other applications and architectures, but this approach has the potential to provide much needed information to the multicomputer user community. Performance prediction tools can aid multicomputer users and designers in increasing parallel efficiency on these machines. High efficiency parallel execution will be essential if "Grand Challenge" problems are to be solved on multicomputers.

References

[1] M. Annaratone and R. Ruhl. Balancing interprocessor communication and computation on torus-connected multicomputers running compiler-parallelized code. In *Proceedings SHPCC 92*, pages 358–365, March 1992.

[2] V. Balasunderam, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. *SIGPLAN Notices*, 26(7):213 – 223, July 1991.

[3] R. G. Born and J. R. Kenevan. Theoretical performance-based cost-effectiveness of multicomputers. *The Computer Journal*, 35(1):63–70, 1992.

[4] T. Fahringer and H. P. Zima. A static parameter based performance prediction tool for parallel programs. Technical Report ACPC/TR 93-1, University of Vienna Department of Computer Science, January 1993.

[5] H. P. Flatt. Further results using the overhead model for parallel systems. *IBM J. Res. Develop.*, 35:721 – 726, 1991.

[6] H. P. Flatt and K. Kennedy. Performance of parallel processors. *Parallel Computing*, 12:1 – 20, 1989.

[7] A. J. Goldberg and J. L. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, January 1993.

[8] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Stat. Comput.*, 9(4):609–639, July 1988.

[9] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers.* The MIT Press, Cambridge, Massachusetts, 1991.

[10] R. W. Hockney and E. A. Carmona. Comparison of communications on the intel ipsc/860 and touchstone delta. *Parallel Computing*, 18(9):1067 – 1072, 1992.

[11] Intel Corporation. *Paragon OSF/1 C Compiler User's Guide*, January 1993.

[12] A. Kapelnikov, R. R. Muntz, and M. D. Ercegovac. A methodology for performance analysis of parallel computations with looping constructs. *Journal of Parallel and Distributed Computing*, 14(2):105–120, February 1992.

[13] D. McCallum and M. J. Quinn. A graphical user interface for data-parallel programming. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, pages 5–13. IEEE Computer Society Press, 1993.

[14] D. Morris and D. Evans. Modelling distributed and parallel computer systems. *Parallel Computing*, 18(7):793–806, July 1992.

[15] S. A. Moyer. Performance of the ipsc/860 node architecture. Technical Report IPC-TR-91-007, Instidute for Parallel Computation, School of Engineering and Applied Science, University of Virginia, May 17, 1991.

[16] D. Muller-Wichards. Problem size scaling in the presence of parallel overhead. *Parallel Computing*, 17(12):1361 – 1376, December 1991.

[17] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. Technical Report UCB/CSD 92/#675, Computer Science Division – EECS, University of California, Berkeley, CA 94720, March 1992.

[18] X. Zhang. Performance measurement and modeling to evaluate various effects on a shared memory multiprocessor. *IEEE Trans. Software Engineering*, 17(1):87 – 93, January 1991.