Appeared in "Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems

October, 1994"

Architectural Scaling and Analytical Performance Prediction †

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

Abstract

Recent advances in the power of parallel computers have made them attractive for solving large computational problems. High speed parallel processors ranging from clusters of workstations to Massively Parallel Processing (MPP) machines are becoming increasingly available to the scientific community [3]. In this paper, the term "scalability" is used to indicate how well an application is able to utilize a varying number of processors. This research develops an analytical model which is used to predict the scalability of data-parallel programs as the parallel architecture varies. This model can be used to determine the portability of applications as well as the impact of architectural features on performance.

1 Introduction

One of the most important advances in high performance computing is the increasing availability of commercial multicomputers. For our purposes, multicomputers will be defined as parallel computers where the processors do not have shared memory [14]. This definition includes everything from clusters of workstations on Ethernet to machines with much faster and more complex interconnection schemes. These systems promise to provide solutions to many problems that require more computational resources than are available on conventional sequential processors.

The current high performance computing environment has changed significantly with the advent of high speed workstations. Much of the work that was previously performed on large centralized computers is now done on a desktop workstation. Clusters of workstations are often available for larger jobs with Massively Parallel Processing (MPP) machines being reserved for the most demanding applications (see Figure 1). We will use the term "Architectural Scaling" to denote the process of writing an application for platforms ranging from a cluster of workstations to a MPP system.

† This research was supported by Intel Foundation and NSF grant ASC-9208971.

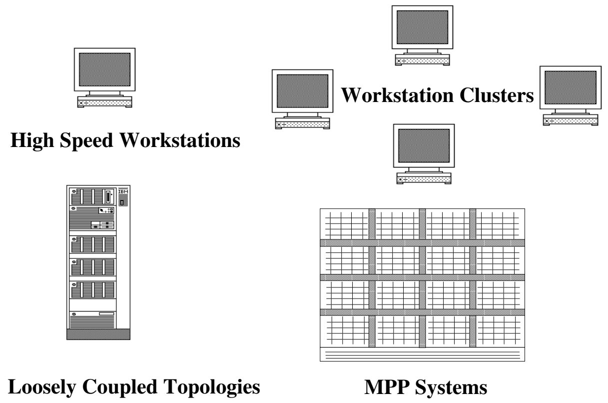

Figure 1: Diverse architectures available to a single researcher.

A developer will often find that an application which has been written for one specific architecture will exhibit poor performance on the next generation of parallel machines. Applications which have been tuned for acceptable performance across a broader range of platforms will be able to more effectively exploit available computing resources.

Two problems have been noted that cause reported performance results to be much poorer than might be expected when actual programs are developed to run on multicomputers:

- It is difficult for a programmer to predict the effect that changes in the source code will have on actual performance. There is also little information available to indicate which parallel architecture, if any, will execute a given program in an efficient manner. In order to be useful, a performance tool must not require the programmer to rewrite the application and must generate results quickly and accurately.

- Parallel system designers are handicapped by the lack of information on the effect changing system parameters will have on the performance of actual parallel programs. As a result, systems may be designed with such a high level of imbalance that few parallel programs will be able to run efficiently on them. Performance tools for use in solving this problem must allow an architect to examine performance for real applications without needing sample runs on the target machine.

Historically there have been two significant barriers to the development of accurate performance models for multicomputers. Programmers who are familiar with the structure of parallel applications often do not have the motivation or hardware background to build accurate models of parallel architectures. As a result, they usually do not have adequate performance feedback during the development of a parallel application. Secondly, architects who understand the underlying systems have not had access to algorithmic information from real applications to use as input into their models. Consequently, engineering models typically use statistical inputs or simplified program "kernels" instead of data collected from the execution of non-trivial parallel programs. As a result there has not been a consistent improvement in the balance between the subsystems of multicomputers [5, 10, 16]. The modeling technique described here provides a vital bridge between parallel system designers and programmers. It allows programmers to evaluate algorithmic choices on different parallel architectures, and it allows system designers to understand the effects of varying system parameters using real application data.

High level parallel languages are essential in making parallel processing 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 [9] programming language provides a SIMD model of parallel programming with explicit parallel extensions to the C language. Our model uses algorithmic information extracted from the source code of an existing Dataparallel C application to provide performance predictions across different hardware platforms. The concepts developed here can also be extended to other data-parallel languages.

Although scaled speedup can be used to examine the efficiency of some scientific applications, many "Grand Challenge" problems require drastic reductions in execution time before a computational solution will be practical. For this reason, our research has concentrated on looking at the relative speedup attained on a fixed problem size as the principle performance metric.

This research presents and evaluates an analytical model designed to support Architectural Scaling and analyzes the success of the model in predicting the performance of parallel algorithms on different hardware platforms. Additional details are available in a related technical report [4]. This kind of performance prediction information should enable parallel programs and system architects to improve the performance of multicomputers.

2 Model Development

Several trends in modern distributed memory parallel systems allow us to use a simplified model for performance prediction. The time to execute the program on a single processor $T_1 = T_s + T_p$ where T_s is the inherently sequential part of the algorithm and T_p is the parallelizable part of the algorithm. Previous research [8, 13] has suggested the following formula to model the parallel execution time of an algorithm:

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

where $\sigma(p, \tau)$ is a function which estimates the communication overhead given the topology τ. We use an enhanced version of this formula in this research.

2.1 Native Compiler Effects

The Dataparallel C compiler 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_ϕ 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.

2.2 Virtual Processor Overhead

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 (VP) emulation. The number of times the compiler must set up a VP emulation loop ($\mathcal{X}_\lambda$) can be used to estimate the parallelization overhead in the computation. The time spent in parallel overhead, T_λ, will be accounted for in our model by the term $C_\phi \mathcal{X}_\lambda$.

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

$$S(p) = \frac{T_1}{\tau(p)} = \frac{T_s + T_p}{T_s + \frac{T_p}{p} + \sigma(p, \tau) + T_\lambda}$$

This reduces to Amdahl's law when $\sigma(p,\tau) = 0$ and $T_\lambda = 0$.

2.3 Floating Point Performance

In the past, floating point arithmetic was so much more time consuming than integer arithmetic that integer instructions were 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 (SP1) 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, the parse tree generated by the compiler can be examined in order to determine the number of operations in sequential code ($\mathcal{X}_s$) and in parallel code ($\mathcal{X}_p$).

2.4 Communication overhead

The $\sigma(p,\tau)$ 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 $\sigma(p,\tau)$ 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_c). 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 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 has increased the overhead associated with starting a communication. Applications which transmit large data sets will also see the the network bandwidth as a first order effect, but for problems with neighbor communications it can often safely be ignored. The number of communications $\mathcal{X}_c$ can be determined at compile time. We will define the normalized message startup cost C_c as the ratio T_c/T_ϕ (where T_ϕ is the time to execute a floating point instruction). The model will estimate the total number of cycles spent in communication to be $\mathcal{X}_c C_c$.

2.5 Cache effects

References to parallel variables in Dataparallel C are grouped into arrays of structures with an element for each virtual processor. When a program enters parallel mode, we assume that the entire array containing parallel variables will be accessed. This assumption holds for the problems we will be studying in this research and is approximately true for many other irregular computational problems. Because of the predictable sequential access of these arrays, it is possible to predict the number of cache misses which will occur. Cache behavior can have a significant effect on the performance of popular multicomputer processors including the iPSC/860 [12]. The number of uncached memory accesses ($\mathcal{X}_m$) and the number of cycles necessary to access an uncached memory location (C_m) account for the time spent waiting for cache lines to fill.

2.6 Model Formulation

Using information extracted from the source code, the compiler can estimate $\mathcal{X}_s$ and $\mathcal{X}_p$, the number of operations in the sequential and parallelizable portions of the code. Let $T_s = C_\phi \mathcal{X}_s T_\phi$ and $T_p = (C_\phi \mathcal{X}_p + C_m \mathcal{X}_m) T_\phi$.

We can express $\sigma'(p,\tau) = \sigma(p,\tau)/T_\phi$ in terms of the normalized startup time. For a broadcast communication on a hypercube topology $\sigma'(p,\tau) = \mathcal{X}_c C_c(1+\log(p))$. Neighbor communications would result in $\sigma'(p,\tau) = \mathcal{X}_c C_c$

Using the dominant effects we have described here,

$$
\begin{aligned}
T_s &= (C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m \mathcal{X}_m) T_\phi \\
T_p &= C_\phi \mathcal{X}_s T_\phi + \frac{(C_\phi \mathcal{X}_p + C_m \mathcal{X}_m) T_\phi}{p} + T_\phi \sigma'(p,\tau) + \\
&\quad T_\phi C_\phi \mathcal{X}_\lambda
\end{aligned}
$$

With speedup

$$
\begin{aligned}
S(p) &= T_s/T_p \\
&= \frac{C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m \mathcal{X}_m}{C_\phi \mathcal{X}_s + \frac{C_\phi \mathcal{X}_p + C_m \mathcal{X}_m}{p} + \sigma'(p,\tau) + C_\phi \mathcal{X}_\lambda}
\end{aligned}
$$

the T_ϕ terms drop out and we are left with a speedup equation dependent only on the variables which are available to our prediction model.

The terms $\mathcal{X}_s$, $\mathcal{X}_p$, $\mathcal{X}_m$, $\mathcal{X}_\lambda$ and $\mathcal{X}_c$ can all be determined from the internal parse tree generated by the compiler. The term C_ϕ can be determined through a sample program or through experience with the compiler on other processors. C_ϕ describes the efficiency of the compiler in generating optimized code. The terms C_c and C_m can be obtained from system specifications or through benchmark programs.

Given an equation for $S(p)$, we can derive expressions describing the levels of balance which are required for an efficient parallel architecture. For our analysis we will assume that an efficient execution of an algorithm will require a relative speedup of at least $p/2$ where p processors are used. This results in processor utilization of 50%.

Theorem 2.1 *In order to achieve a utilization of 50% on grid problems with neighbor communications, the following inequality must hold.*

$$C_c < \frac{C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m\mathcal{X}_m - (p-1)C_\phi\mathcal{X}_s - pC_\phi\mathcal{X}_\lambda}{p\mathcal{X}_c}$$

Proof:

Given $S(p)$, we can determine the utilization

$$\begin{aligned} \mathcal{U}(p) &= \frac{S(p)}{p} \\ &= \frac{1}{1 + \frac{(p-1)C_\phi\mathcal{X}_s + p\sigma'(p,\tau) + pC_\phi\mathcal{X}_\lambda}{C_\phi(\mathcal{X}_s+\mathcal{X}_p)+C_m\mathcal{X}_m}} \end{aligned}$$

In order to achieve a utilization greater than 50%

$$\frac{(p-1)C_\phi\mathcal{X}_s + p\mathcal{X}_cC_c + pC_\phi\mathcal{X}_\lambda}{C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m\mathcal{X}_m} < 1$$

This can only occur when

$$C_c \quad < \quad \frac{C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m\mathcal{X}_m - (p-1)C_\phi\mathcal{X}_s - pC_\phi\mathcal{X}_\lambda}{p\mathcal{X}_c}$$

□

Several intuitive relationships can be observed as special cases of Theorem 2.1. If the message startup cost, C_c, is to have a positive value, then:

$$C_\phi(\mathcal{X}_s + \mathcal{X}_p) + C_m\mathcal{X}_m > (p-1)C_\phi\mathcal{X}_s - pC_\phi\mathcal{X}_\lambda \qquad (1)$$

If we assume that the memory component of the execution time ($C_m\mathcal{X}_m$) and the loop overhead ($pC_\phi\mathcal{X}_\lambda$) are not dominant, then Equation 1 reduces to $\mathcal{X}_p > p\mathcal{X}_s$. The parallel component of a computation must be more than p times as large as the sequential component if $\mathcal{U}(p)$ is to be greater than 50%.

Assuming memory accesses and loop overhead are insignificant and that the parallel component of the execution time dominates the sequential time

$$C_c < \frac{C_\phi\mathcal{X}_p}{p\mathcal{X}_c}$$

Essentially, this expression indicates that the grain size must be larger than the cost of a communication.

We can also observe that higher communication costs can be tolerated when a slower memory subsystem is used or when a poor compiler causes increased values of C_ϕ. Theorem 2.1 also indicates the sensitivity of parallel computations to VP emulation loop overhead.

3 Experimental Results

Several experiments were performed to validate the analytical model. One set of experiments was performed to determine if the model 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 model 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.

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 implementation from parallel local variables to parallel global variables. The experiments were performed on the Intel iPSC/860, iPSC/2 and the iWarp array, Our model was able to accurately

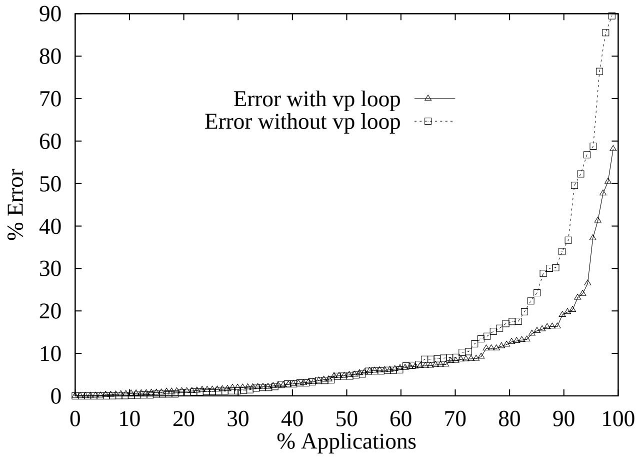

Figure 2: Percent error for speedup on 107 experimental runs of the matrix multiplication, shallow water, ocean circulation and sharks world applications. The experiments were performed on the Intel iWarp array, the iPSC/860 and the iPSC/2.

determine that the use of global loop variables improved performance.

We also tested our model using a shallow water application developed by the National Center for Atmospheric Research, an ocean circulation model, and a sharks world application which simulates the interaction between sharks and fish in a toroidal ocean [9]. For our analysis we have used the classical error computation method:

$$Error = \frac{\text{Predicted speedup} - \text{Actual speedup}}{\text{Actual speedup}}$$

The error contour shown in Figure 2 indicates that over 80% of the experimental runs achieve less than 10% error. Figure 2 also shows that a model incorporating the term for virtual processor overhead achieves significantly lower error rates.

Although machines with extremely low latency memory and network subsystems may promise higher performance, a more inexpensive solution may be acceptable for many users. If we assume an exponential increase in cost, corresponding to improvements in memory and network speed, a plot of performance per dollar can be generated as is shown in Figure 3. Given this information, manufacturers can position new machines near the cost optimal points for important applications. Users can also select parallel architectures which will be most economical for their particular needs. Results from this analytical model can be of significant use to systems architects and MPP users.

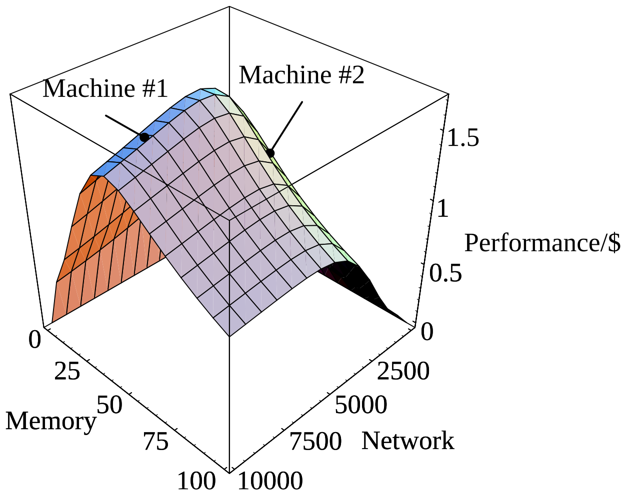

Figure 3: Predicting MFlops/dollar for the Shallow Water Model

Modeling algorithmic behavior can help system designers know what effect their design decisions will have on the efficiency of an algorithm family.

4 Related Work

Several different approaches have been taken in modeling parallel systems and predicting performance for real applications. Balasundaram et al. [2] have developed a performance estimator which matches sections of source code to templates in a training set. Annaratone has developed a tool that uses the communication/computation ratio to make decisions in parallelizing Fortran code [1]. A static parameter based prediction tool was developed by Fahringer [7] to guide the compiler in optimization decisions. Crovella describes a performance predicates [6] approach which uses data collected at run time to determine bottlenecks for an application.

The principle difference between our model and other research is that it allows for performance prediction across diverse architectures for which there is no available implementation. It is important for system architects to be able to predict performance on future machines. Architectural scaling is an important area where performance prediction for a continuum of architectures must be available. With this prediction methodology programmers can estimate how much performance tuning will be necessary to achieve efficient execution on a variety of parallel architectures.

5 Summary

At the Pasadena Workshop [15], High Performance Computing (HPC) users met to define key software technologies necessary to improve the parallel processing environment. The three goals they specified for system software and tools were performance, portability and usability. Performance prediction models are an important element in achieving all three of these goals. The analytical model described in this paper allows users to increase the performance of applications across the spectrum of available multicomputer architectures. The portability of an application can be determined through visualizing performance as architectural features vary. Since the information used by this model is obtainable from the source code, the usability of this technique is also high. The use of predicted performance in the process of architectural scaling can allow programmers and system architects to improve the efficiency of applications executing 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*, pp. 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*, vol. 26, pp. 213 – 223, July 1991.

[3] G. Bell, "Ultracomputers, a teraflop before its time," *Communications of the ACM*, vol. 35, pp. 27–47, August 1992.

[4] M. J. Clement and M. J. Quinn, "Using analytical performance prediction for architectural scaling," Tech. Rep. 94-80-10, Department of Computer Science, Oregon State University, July 1994.

[5] M. J. Clement, M. J. Quinn, and B. Baxter, "Medium grain size applications on distributed memory multicomputers," Tech. Rep. 93-80-13, Department of Computer Science, Oregon State University, September 1993.

[6] M. Crovella, R. Vianchini, T. LeBlanc, E. Markatos, and R. Wisniewski, "Using communication-to-computation ratio in parallel program design and performance prediction," in *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pp. 238–245, December 1992.

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

[8] H. P. Flatt and K. Kennedy, "Performance of parallel processors," *Parallel Computing*, vol. 12, pp. 1 – 20, 1989.

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

[10] R. W. Hockney and E. A. Carmona, "Comparison of communications on the intel iPSC/860 and touchstone delta," *Parallel Computing*, vol. 18, no. 9, pp. 1067 – 1072, 1992.

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

[12] S. A. Moyer, "Performance of the iPSC/860 node architecture," Tech. Rep. IPC-TR-91-007, Institute for Parallel Computation, School of Engineering and Applied Science, University of Virginia, May 17, 1991.

[13] D. Muller-Wichards, "Problem size scaling in the presence of parallel overhead," *Parallel Computing*, vol. 17, pp. 1361 – 1376, December 1991.

[14] M. J. Quinn, *Parallel Computing: Theory and Practice*. New York, New York: McGraw-Hill Book Company, 1994.

[15] P. H. Smith, "System software and tools for high performance computing environments." Report on the Findings of the Pasadena Workshop, April 14-16 1992.

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