Appeared in "Computer Systems Science and Engineering 1995"

Using Analytical Performance Prediction for Architectural Scaling †

Mark J. Clement
Computer Science Department
Brigham Young University
Provo, Utah 84602
clement@cs.byu.edu

Michael J. Quinn
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202
quinn@research.cs.orst.edu

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 [4]. 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 [22]. 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.

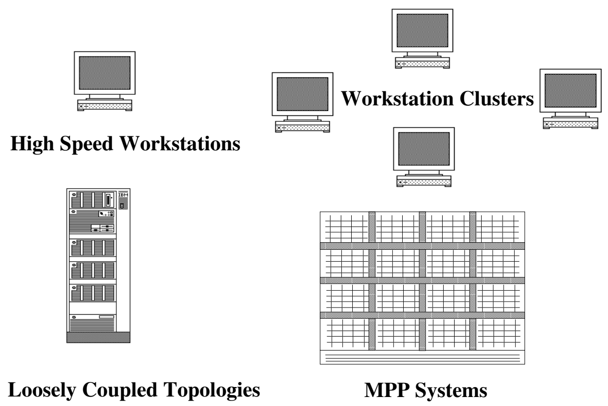

Figure 1: Diverse architectures available to a single researcher.

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.

Without an accurate model, it is difficult to determine what changes are needed to improve the performance of multicomputers. Performance prediction tools have been identified as an important technology in improving performance for grand challenge problems [8].

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 [7, 15, 25]. 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 [14] 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.

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. This kind of performance prediction information should enable parallel programs and system architects to improve the performance of multicomputers.

2 Motivation

Performance prediction information is useful only if it can be obtained at a reasonable cost in terms of programmer effort and computational resources. This model has been tailored for the following uses.

- 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. For this reason, our performance model will rely primarily on algorithmic information extracted from the source code.

  Where multiple parallel architectures are available, performance prediction information can also allow the user of an application to choose the architecture and number of processors which will result in the most efficient use of computational resources. Because our model can make predictions of MPP performance from data collected on smaller systems, it offloads performance tuning tasks from large production systems. This is particularly valuable when a smaller configuration of a multicomputer is locally available and the larger system is a shared national resource.

- 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 the available processors. 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. Speedup is often used as a measure of how efficient an architecture is at executing a parallel algorithm. Although opinions are divided on exactly how speedup should be measured, there is some evidence that a majority of computational oriented mathematical researchers feel that speedup should be used as the primary measure of parallel performance [3]. The speedup achieved by an algorithm running on p processors can be defined as:

$$Speedup(p) = \frac{\begin{array}{c}\textit{Time to solve a problem using the best}\\ \textit{sequential algorithm on one processor}\end{array}}{\begin{array}{c}\textit{Time to solve the same problem with}\\ \textit{the parallel algorithm on p processors}\end{array}}$$

It is often extremely difficult to determine the execution time of the best sequential algorithm for a given problem. For this reason we will use parallelizability, or relative speedup, as the primary performance measurement in this research. Relative speedup is defined as:

$$S(p) = \frac{\begin{array}{c}\textit{Time to solve a problem using the parallel}\\ \textit{algorithm on one processor}\end{array}}{\begin{array}{c}\textit{Time to solve the same problem with}\\ \textit{the parallel algorithm on p processors}\end{array}}$$

Some research has suggested that scaled speedup is an important metric to use in evaluating parallel performance [13]. 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 [11]. 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 relative speedup attained on a fixed problem size as the principle performance metric.

4 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 [12, 21] 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.

4.1 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.

4.2 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 $T_\lambda = 0$.

4.3 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 [16]. 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 ($\mathcal{X}_p$).

4.4 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 $\mathcal{X}_c C_c$.

4.5 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 [20]. 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. Using these parameters, the performance prediction model can estimate the number of cycles spent waiting for uncached memory.

4.6 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,

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

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 4.1 *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 4.1. 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 4.1 also indicates the sensitivity of parallel computations to VP emulation loop overhead.

5 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.

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 [14]. 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 experiments were performed on the Intel iPSC/860, iPSC/2 and the 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. Our model was able to accurately predict the performance of the original

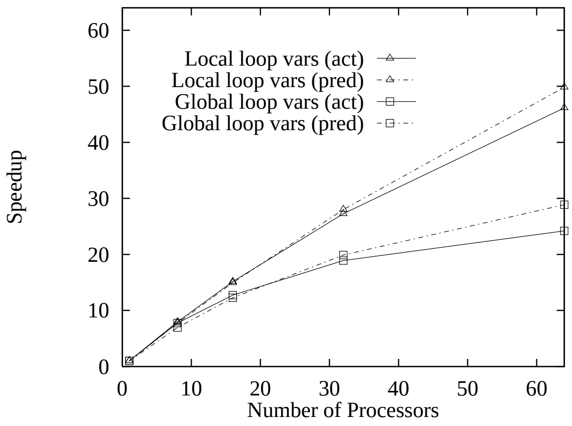

Figure 2: Experimental and predicted values results for 256x256 matrix multiplication on the iWarp array.

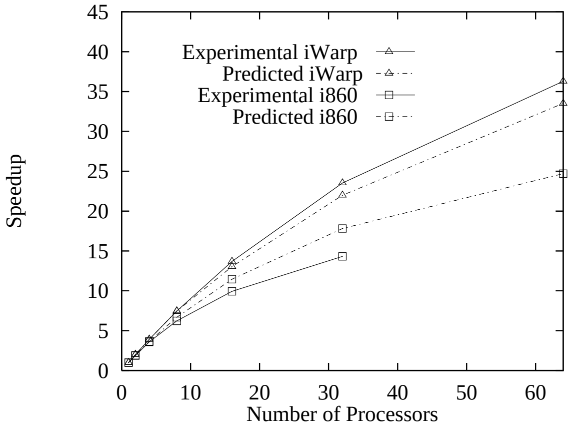

Figure 3: Experimental and predicted results for the shallow water atmospheric model.

version and the new version called "matrix2+" . The results are shown in Figure 2.

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 several standard Dataparallel C applications and were reported previously [6]. The experimental results indicate that the analytical model is successful in predicting performance in the cases we have examined. Given an accurate model, important analysis can occur with respect to a specific application.

The shallow water atmospheric application was developed by the National Center for Atmospheric Research for benchmarking the performance of parallel processors [14]. 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 model is able to approximate the actual performance fairly accurately. More significantly, the model was able to differentiate clearly between the performance to be expected on the two machines. The results are shown in Figure 3.

Performance information from an analytical model can allow a system architect to observe the effects of changing specific system parameters. In Figure 5 the message startup cost is varied for the shallow water atmospheric model to show the effect this parameter has on speedup. Figure 6 shows how cache miss penalty and message startup cost interact in predicted speedup results.

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 4. 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.

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

We also tested our model using an ocean circulation model, and a sharks world application which simulates the interaction between sharks and fish in a toroidal ocean. 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 8 indicates that over 80% of the experimental runs achieve less than 10% error. Figure 8 also shows that a model incorporating the term for virtual processor overhead achieves significantly lower error rates.

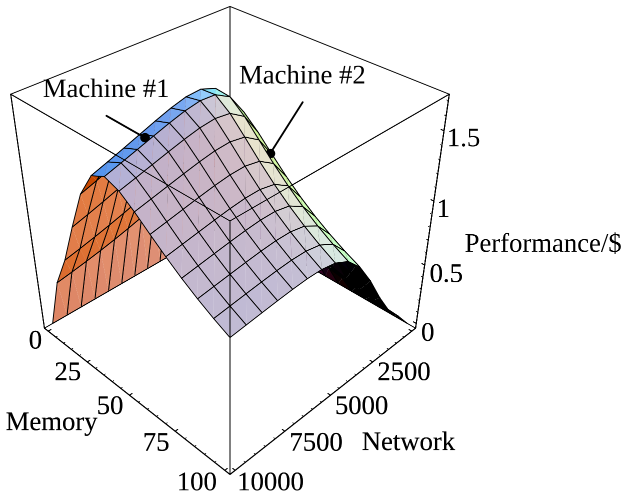

Figure 4: Predicting MFlops/dollar for the Shallow Water Model

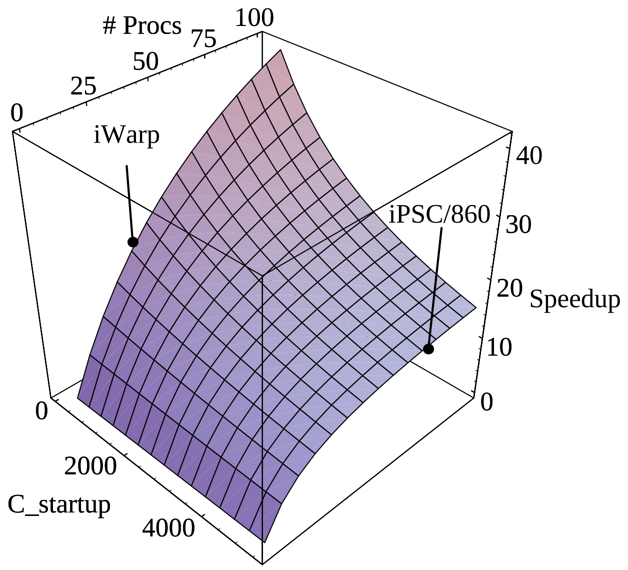

Figure 5: Speedup results for the shallow water atmospheric model with variable message startup cost. C_startup is the message startup time divided by the time for an arithmetic operation. A C_startup value of 470 corresponds to the iWarp processor and a value of 5000 approximates the iPSC/860.

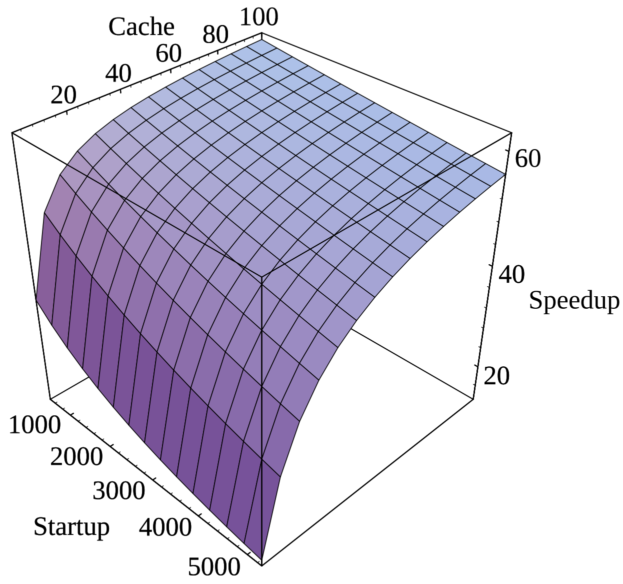

Figure 6: Speedup results for the shallow water atmospheric model for 64 processors with variable message startup cost and cache miss penalty.

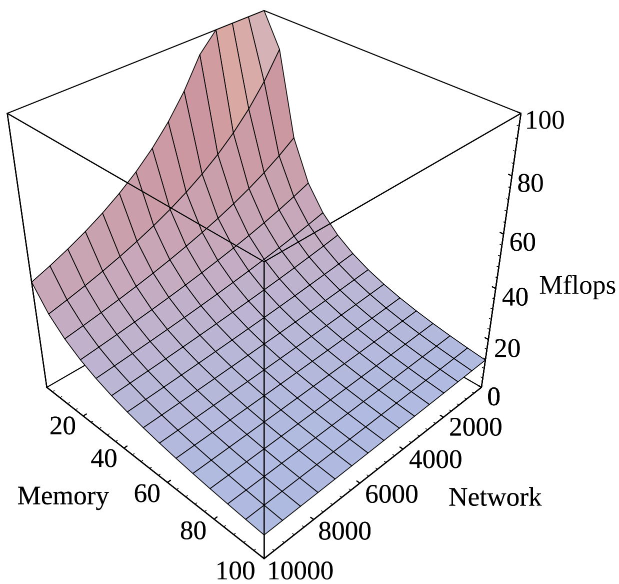

Figure 7: Mflops results for the shallow water atmospheric model for 64 processors and variable message startup cost and cache miss penalty. A processor speed of 10Mflops is assumed for each processing element.

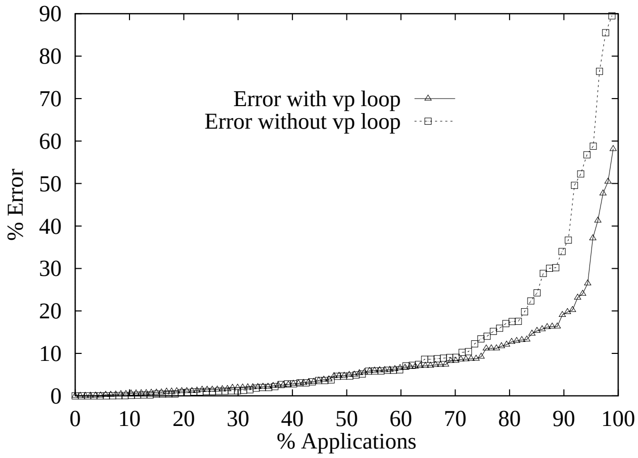

Figure 8: Percent error for speedup on 107 experimental runs of the matrix multiplication, shallow water, ocean circulation and sharks world applications. The experiments were performed on the Intel iWarp array, the iPSC/860 and the iPSC/2.

5.3 External Sorting

External sorting is a disk intensive problem. N records must be read and written to disk while $O(Nlog(N))$ comparison operations are performed. Additionally, in the worst case, all N records will have to pass through the bisectional bandwidth of the interconnection network. If the disk speed is not well balanced with the processor and network speed, the time spent reading and writing data will often dominate [18]. There are few guidelines as to how well balanced these system parameters should be in order to achieve acceptable performance. Preliminary work has been done to predict the effects of architectural features on the sorting problem. We have developed a new parallel sorting algorithm that maximizes the overlap between the disk, network and CPU subsystems on a parallel node [5]. This algorithm hides much of the imbalance of a particular subsystem behind the operations of the others. Because of this overlap, we are able to draw some conclusions about the minimum levels of system balance that are necessary for any sorting algorithm. A model was built using this algorithm to examine the sorting problem on parallel architectures.

Figure 9 shows the speedup values predicted on 100 processors with skew equal to 1.5 as the network and disk speeds vary. The system parameters of several actual machine architectures have been labeled to show the estimated performance of the overlapped sorting algorithm on actual machines. All of the machine plots were made assuming SCSI disks on each node with a 3Mbyte/sec transfer speed. Figure 10 shows a close up view of the region of maximum speedup. The Intel Paragon is shown to be on the plateau of maximum performance, but it is near the point where severe performance degradation would occur if network transmission time were increased. The network on the Paragon was assumed to have 200Mbyte/sec links configured in a mesh [16]. The IBM SP1 is plotted assuming a 6Mbyte/sec omega network [17]. The Meiko CS2 has a logarithmic network where bisectional bandwidth increases linearly with the number of processors. The Meiko plot assumes that each node has a 50Mbytes/second link [19]. The SP1 appears to have higher disk transmission times because of the faster CPU speed. Both the network and disk axes are given in terms of the CPU speed. The Ethernet plot in Figure 4 pertains to a network of SPARC workstations with SCSI disks connected with Ethernet. It is obvious that the network bisectional bandwidth must be significantly increased before a network of workstations can be used with the overlapped sorting algorithm.

In order to validate our analytical sorting model, we used algorithmic information from the *Parallel Sorting by Regular Sampling* (PSRS) [23] algorithm in our model. We then compared our speedup predictions to those documented by the authors of the PSRS paper. [23] Our results were within 10 percent of the empirical results for the iPSC/2 and the iPSC/860 multicomputers. The authors claimed 37% lower performance than we predicted for the network of 8 Sun 3/80 workstations. It was impossible to determine the exact test environment used with the workstations and we attribute the additional error to our assumptions. Table 1 shows the predicted and actual results for the PSRS algorithm. In Figure 11 we show the impact of message startup time on the PSRS algorithm. Before the last merging phase of the algorithm can proceed, p communications must be initiated by each processor. As the number of processors becomes large, this message startup time degrades the speedup attained. With the analytical model we have proposed, algorithmic limitations can be detected without using critical resources on MPP systems. The majority of time spent in performance debugging should be spent working with a performance prediction tool in a workstation environment.

Proper system balance is important to achieving acceptable speedup on the sorting problem. Modeling algorithmic behavior can help system designers know what effect their design decisions will have on the efficiency of an algorithm family.

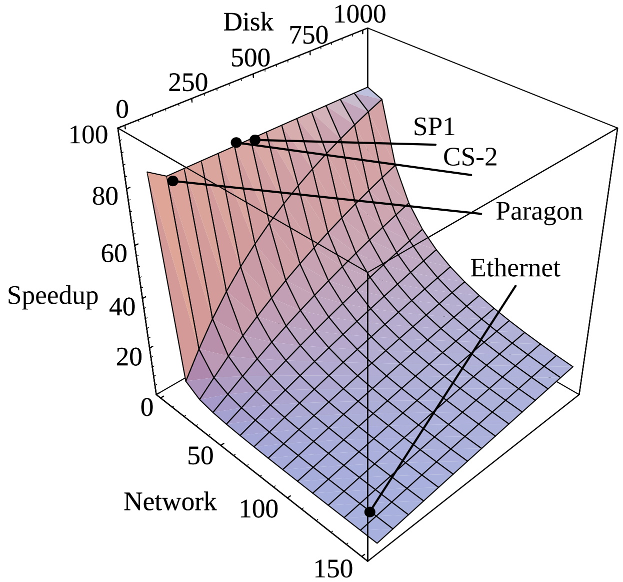

Figure 9: Results of an analytical model developed to investigate the impact on the sorting algorithm of varying system parameters. The Disk and Network axis are given in number of comparison times to transfer a tuple. Skew is assumed to be 1.5. Points are plotted for the Intel Paragon, IBM SP1, Meiko CS2 and a network of SPARC workstations connected by Ethernet.

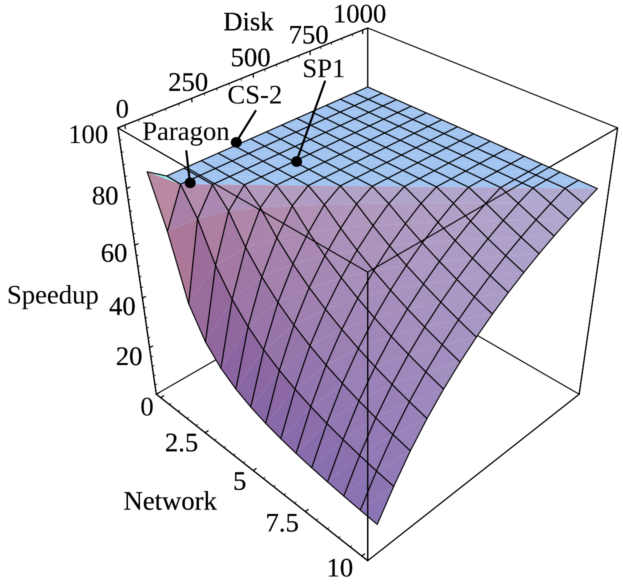

Figure 10: Closer view of the maximum speedup plane of the analytical model. The Disk and Network axis are given in number of comparison times to transfer a tuple. Skew is assumed to be 1.5. Points are plotted for the Intel Paragon, IBM SP1 and Meiko CS2.

Machine	Nodes	Actual Speedup	Predicted Speedup	Error
iPSC/2-386	32	27.5	25.0	9.08%
iPSC/860	64	38.0	34.0	10.6%
Network	8	4.0	5.5	37.1%

Table 1: Actual and Predicted speedups for the PSRS algorithm. Actual speedup results from [23].

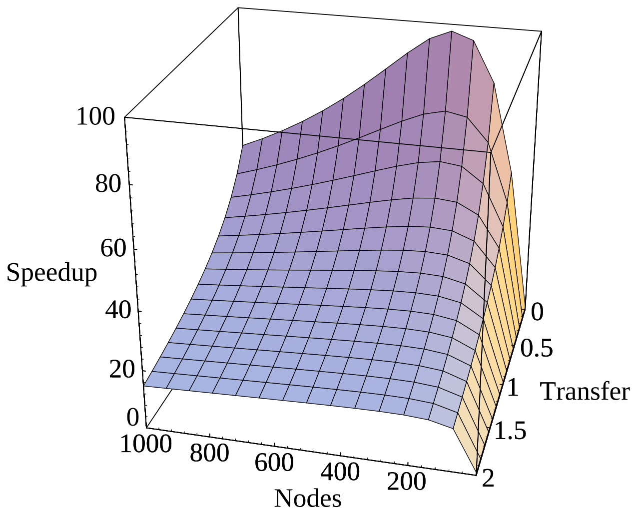

Figure 11: Analytical results for the PSRS algorithm as the number of processors becomes large. The transfer time is given in number of comparison times to transfer an integer. The message startup time is set at 5000 comparison times. This corresponds to the startup time for the iPSC/860.

6 Related Work

Several different approaches have been taken in modeling parallel systems and predicting performance. Most 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. None of the tools are suited to predicting the performance effects of architectural scaling.

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. The training set approach combines several of the effects that must be separated to predict the effects of architectural scaling. The training set approach also does not account for the cache miss penalty, which is important in the class of processors which we are studying here.

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. It does not extend well into an environment where performance is to be predicted on non-existent architectures.

A static parameter based prediction tool was developed by Fahringer [10]. 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. This tool is then used to guide the compiler in optimization decisions.

Crovella describes a performance prediction tool for shared memory and NUMA architectures [9]. Their tool is tailored specifically to provide performance tuning information and describes the importance of simplifying the model. Their performance predicates approach to performance analysis results in lower levels of perturbation than other trace driven analysis.

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.

7 Summary

At the Pasadena Workshop [24], 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] R. S. Barr and B. L. Hickman, "Reporting computational experiments with parallel algorithms: Issues, measures, and experts' opinions," *ORSA Journal on Computing*, vol. 5, pp. 2–18, Winter 1993.

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

[5] M. J. Clement and M. J. Quinn, "Overlapping computations, communications and I/O in parallel sorting." In Revision for *Journal of Parallel and Distributed Computing*, June 1994.

[6] M. J. Clement and M. J. Quinn, "Analytical performance prediction on multicomputers," in *Proceedings of Supercomputing '93*, pp. 886–905, November 1993.

[7] 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.

[8] Committee on Physical, Mathematical, and Engineering Sciences Federal Coordinating Council for Science, Engineering, and Technology, *Grand Challenges 1993: High Performance Computing and Communications*. Washington, D.C.: National Science Foundation, 1993.

[9] 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.

[10] 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.

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

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

[13] J. L. Gustafson, G. R. Montry, and R. E. Benner, "Development of parallel methods for a 1024-processor hypercube," *SIAM J. Sci. Stat. Comput.*, vol. 9, pp. 609–639, July 1988.

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

[15] 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.

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

[17] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures*. San Mateo, California: Morgan Kaufmann Publishers, Inc., 1992.

[18] C. Leopold, "A fast sort using parallelism within memory," in *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pp. 326–333, 1992.

[19] Meiko World Incorporated, *Computing Surface 2 Overview Documentation Set*, 1993.

[20] 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.

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

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

[23] H. Shi and J. Schaeffer, "Parallel sorting by regular sampling," *Journal of Parallel and Distributed Computing*, vol. 14, pp. 361–372, April 1992.

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

[25] 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.