Multiprocessor Scalability Predictions Through Detailed Program Execution Analysis *

Xiaodong Zhang Zhichen Xu

High Performance Computing and Software Laboratory
The University of Texas at San Antonio
San Antonio, Texas 78249

Abstract

Scalability measures the ability of a parallel system to improve performance as the size of an application problem and the number of processors involved increase. There are some limits to existing scalability studies. First, the problem size in a computation is not well-defined. Second, the methods used to differentiate algorithmic and architectural scalabilities are not effective enough. Thirdly, most approaches to scalability study are either highly time-consuming or restricted to simple problem/architecture structures. A major effort of this work is to address these limits. We have extended the latency metric [11] for more complex scaling of problems, and to possibly isolate the scalability of an algorithm from a parallel system. The scalability prediction is based on a semi-empirical approach [10] that significantly reduces the time and cost of measurements and simulation. Our prediction results were validated on the KSR-1 and on the CM-5.

1 Introduction

There are some limits to existing scalability studies. First, most scalability studies consider the input data size of a problem, or the number of floating point operations in the computation as the problem size. In practice, the growth and scaling of an application problem is more complex. In many large scientific simulations of physical phenomena, more than one parameter is used to change the input data size and the number of floating point operations of the programs. The ways of changing multiple input parameters can significantly change the communication and synchronization structure and parallelism of a program even though the resulting data size or the number of floating point operations are the same. This means the scaling behavior of an application will depend on the way the input parameters are scaled. Second, it is important to study program scalability and architecture scalability as independently as possible. Several proposals have been made to separate the two scalabilities. In [5], algorithm scalability is defined as the maximum achievable speedup on an architecture with an idealized communication structure. This speedup measures the inherent parallelism and overhead patterns of a program for a given size. To isolate architectural effects, architecture scalability for a program is defined as the ratio of the speedup of the program on a real machine and the asymptotic speedup of the program on an EREW PRAM. An analytical model of a program structure may not precisely and completely describe overhead patterns of the program. The architectural scalability is still, to a great extent, dependent on the application program executed. In [12], the execution overhead latency is further divided into the memory reference latency, the processor idle time, and the parallel primitive execution overhead time, which comprehensively cover the major sources of system/architecture overheads. However, the same type of latency in this classification may also come from both the program and the machine. It is still difficult to identify and distinguish the performance bottlenecks from the hardware and the algorithm. In [7], parallel computing overheads are classified into algorithmic overhead and interaction overhead, which isolates the overhead from the application program. However, it does not provide direct information about how performance changes when the size of the application and size of the machine changes. In addition, the simulation-based approach could be highly computation intensive. The third limit is the lack of effective evaluation methodologies to obtain scalabilities analytically and experimentally. Experimental and simulation methods are highly time consuming. Both simulation and analytical methods are based on some assumptions of the architecture and application. Analytical approaches usually describe some asymptotic behavior of a parallel system, and they may only be applicable to simple problem/architecture structures.

To address these limits, we make some extensions to the latency metric in [11], and use a semi-empirical approach for scalability study. The first extension to the latency metric is to isolate the scalability of the algorithm from an algorithm-machine combination. We define the algorithmic scalability of a program based on the definition of algorithm efficiency of the program, which is defined as the ratio of the parallel execution times of the program on two idealized ma-

*This work is supported in part by the National Science Foundation under research grants CCR-9102854, CCR-9400719, and under instrumentation grant DUE 9250265, and by the U.S. Air Force under research agreement FD-204092-64157, and by a grant from the San Antonio Area Foundation Part of the experiments were conducted on the CM-5 machines in Los Alamos National Laboratory and in the National Center for Supercomputing Applications at the University of Illinois, and on the KSR-1 machines at Cornell University and at the University of Washington

ICS '95 Barcelona, Spain © 1995 ACM 0-89791-726-6/95/0007..$3.50

chines: a MIRACLE machine that always exhibits linear speedup regardless of the characteristic of the program, and an idealized PRAM machine with an ideal communication structure. The second extension to the latency metric is related to scaling methods. In this study, we explicitly keep track how the parameters are scaled. To effectively measure and study the various scalabilities, we use a semi-empirical approach that relies on both analytical and experimental methods [10]. This approach has been used to predict the execution times of large application programs on large architectures based on small sample data [9]. There are several advantages of the semi-empirical approach in scalability evaluation. First, execution times on real parallel machines and on simulators used for latency measurements are significantly reduced, because analytical modeling is partially applied, and scalability predictions are based on measurement results of small-scale executions of the programs. Second, a graphical representation of a program enables computation demands of the program to be expressed as functions of application parameters, and all explicit communications to be isolated and classified. Therefore, program overhead patterns are studied according to their underlying semantics and communication structures. Thirdly, the architecture model in our semi-empirical approach is relatively high-level and is less dependent on real architectures. Thus it is more flexible for evaluation on different architectures. Finally, because important and implicit system effects are obtained through experimental measurements, the semi-empirical approach should be more precise than pure analytical models and simulations.

The organization of the paper is as follows. In section 2, we extend the latency metric by quantifying algorithmic scalability of parallel programs and define various scalabilities. Section 3 describes in detail the semi-empirical approach to scalability study. Section 4 presents several case studies on how to use the semi-empirical approach to study the algorithmic scalabilities of various programs and the combination scalabilities on various architectures. The experiments were performed on the KSR-1 and the CM-5. Section 5 concludes the paper.

2 Extensions to the Latency Metric

The latency metric [11] defines algorithm-machine combination scalability as an average increase of the overhead latency (L) needed to keep its computation efficiency equal to a constant (E) when the size of a parallel program increases from W to W', and the size of the parallel architecture increases from P processors to P' processors. It is expressed as

$$scale(E,(P,P')) = \frac{L(W,P)}{L(W',P')}. \quad (1)$$

Using (1), an upper triangular table is generated, where each element represents the scalability measurement of (1) between a meaningful pair of processors. There are three major limitations of the latency metric. First, the experimental measurements are highly time-consuming. Second, the latency measurements require special tools, such as hardware/software monitors, or intensive software instrumentation. Instrumentation overhead can further slow down program execution. Finally, like other scalability metrics, the latency metric gives a single data size as the problem size. To address these limits, we extend the latency metric to quantify algorithmic scalabilities, and to include multiple input parameters to scale programs. Furthermore, scalability measurement time is significantly reduced by our semi-empirical approach.

2.1 Problem Size

The existing definitions of problem size represent the amount of meaningful computation that must be carried out to solve a problem, and the input data size [1, 6, 11], which express computation demand and memory requirement as seen by a conventional serial machine. For a large category of applications that have multiple input parameters, an undesirable phenomenon is that, when we scale the multiple input parameters of an algorithm differently, the resulted performance can be dramatically different. For example, the following program consists of two loops.

```
REAL A(MAX), B(MAX) ;
... ...
A(1:M) = A(1:M) + B(1:M) ;
DO j = 1,N
    B(j) = B(j-1) + A(j) ;
```

Assume that MAX is a large integer number. We set M=0, and N=MAX in the first case, and M=MAX and N=0 in the second case. The numbers of floating point operations and the memory requirements in both cases are equivalent, which means the programs in these two cases have the same problem size. However, the parallel computing performance would be significantly different between the two cases. In the first case, vector A(1:M) will be updated simultaneously by multiprocessors without involving communications. In the second case, the loop to update vector B(1:N) cannot be performed simultaneously because of the dependency in the loop. This example indicates that a program with the same problem size may have different degrees of parallelism which would behave differently in parallel.

To avoid this undesirable situation, we define the size of a problem as the set of all its relevant parameters, and explicitly keep track how the multiple parameters are scaled.

2.2 Algorithm Latency and Efficiency

Factors that limit algorithm scalability are synchronization and communication structures and imbalanced workload of the algorithm. However, remote memory accesses, cache coherence activities and contention for hardware resources will affect the program/machine combination scalability. An ideal machine that excludes all the non-algorithmic factors can be used to isolate the algorithmic latency. Here we define an ideal PRAM in which the cost of a memory access is independent of its access distance, and no contention for hardware resources is presented. To facilitate our presentation, another ideal machine, MIRACLE, is introduced. A MIRACLE machine always exhibits linear speedup regardless of the characteristics of the program being executed. In order to make fair comparisons, the real machine, the MIRACLE machine and the PRAM machine are assumed to be made of the same type of processors.

Let Π denote a parallel program, χ denote the parameters of Π, P denote the number of processors involved in executing the program Π, $T_p^{\mathcal{M}}(\Pi,\chi,P)$ denote the parallel execution time of the program on the MIRACLE machine, $T_p^{\mathcal{P}}(\Pi,\chi,P)$ denote the parallel execution time of Π on the PRAM machine, and $T_p(\Pi,\chi,P)$ denote the parallel execution time of Π on the real machine. We define the average

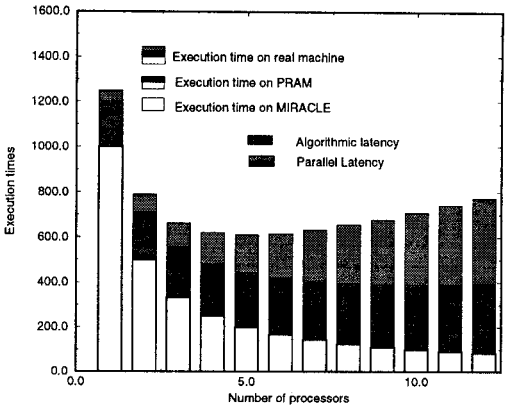

Figure 1: Algorithmic and parallel latencies.

algorithmic latency of Π as

$$\mathcal{L}_a(\Pi, \chi, P) = T_p^{\mathcal{P}}(\Pi, \chi, P) - T_p^{\mathcal{M}}(\Pi, \chi, P), \quad (2)$$

and the average parallel latency as

$$\mathcal{L}(\Pi, \chi, P) = T_p(\Pi, \chi, P) - T_p^{\mathcal{M}}(\Pi, \chi). \quad (3)$$

Figure 1 illustrates a distribution of algorithmic and parallel latencies of a program. Based on the definition of the various latencies, we define algorithmic efficiency as

$$E_a(\Pi, \chi, P) = \frac{T_p^{\mathcal{M}}(\Pi, \chi, P)}{T_p^{\mathcal{P}}(\Pi, \chi, P)}, \quad (4)$$

and parallel efficiency as

$$E(\Pi, \chi, P) = \frac{T_p^{\mathcal{M}}(\Pi, \chi, P)}{T_p(\Pi, \chi, P)}. \quad (5)$$

In (4), E_a measures the percentage of execution time that is dedicated to meanful computation on an ideal machine — hence the efficiency of the algorithm. Since the execution times on a single processor of an ideal communication PRAM, on a single processor of MIRACLE and on a single processor of a real machine are equivalent, we have

$$T_p^{\mathcal{M}}(\Pi, \chi, P) = \frac{T_s^{\mathcal{P}}(\Pi, \chi)}{P}, \quad (6)$$

$$T_p^{\mathcal{M}}(\Pi, \chi, P) = \frac{T_s(\Pi, \chi)}{P}, \quad (7)$$

where $T_s^{\mathcal{P}}(\Pi, \chi)$ and $T_s(\Pi, \chi)$ are the sequential execution times of Π on one node of the PRAM machine and on one node of the real machine, respectively. Substituting Equations (6) and (7) into Equations (4) and (5), we obtain

$$E_a(\Pi, \chi, P) = \frac{T_s^{\mathcal{P}}(\Pi, \chi)}{P T_p^{\mathcal{P}}(\Pi, \chi, P)}.$$

and

$$E(\Pi, \chi, P) = \frac{T_s(\Pi, \chi)}{P T_p(\Pi, \chi, P)}.$$

The definition of efficiencies in this paper coincides with the conventional definition of parallel efficiency. Note that in Equations (2), (3), (4) and (5), the problem size of a program is no longer defined as the number of floating point operations or the input data size but is represented by parameters of the program, χ. The representation of the problem size of a program as application parameters of the problem is more precise and provides more information in our subsequent study of various scalabilities.

2.3 Scalabilities of Parallel Programs and Systems

To extract the scalability of the program from a problem-machine combination, we extend the latency metric by defining algorithmic scalability as the amount of the increase in algorithmic latency to keep a desired algorithmic efficiency, $E_a \in [0, 1]$, that is

$$Scale_a(E_a, (P, P')) = \frac{\mathcal{L}_a(\Pi, \chi, P)}{\mathcal{L}_a(\Pi, \chi', P')}, \quad (8)$$

where P' is a larger number of processors than P, χ' is the scaled application parameter to keep $E_a = E_a(\Pi, \chi', P') = E_a(\Pi, \chi, P)$. The larger the increase of the algorithmic latency to keep a desired algorithmic efficiency, the smaller the algorithmic scalability will be. Smaller algorithmic scalability indicates more extra work involved to parallelize the algorithm. Similarly, we restate the combination scalability of a program-machine combination as

$$Scale(E, (P, P')) = \frac{\mathcal{L}(\Pi, \chi, P)}{\mathcal{L}(\Pi, \chi', P')}, \quad (9)$$

where P' and χ' are the same as in Equation (8), and

$$E = E(\Pi, \chi', P') = E(\Pi, \chi, P).$$

To compare the scalabilities of different algorithms implementing the same problem, we can compare their scalabilities by scaling the application parameters in the same way.

Property 1. *The algorithmic scalability defined in this section is independent of the processor speed of the machines, and thus reflects only the inherent parallelism and workload imbalance of the program.*

Explanation:

According to the definition of the MIRACLE and PRAM Machines, there will be no communication and contention latencies in the execution of any program. For any program Π, $T_p^{\mathcal{M}}(\Pi, \chi, P)$ will be proportional to $\frac{1}{P}$ of the computation demand of Π with parameter χ, that is

$$T_p^{\mathcal{M}}(\Pi, \chi, P) = C\frac{1}{P}\Theta(\Pi, \chi),$$

where C is a processor-related constant, and $\Theta(\Pi, \chi)$ is the computation demand of Π. Similarly, when we represent the program as a thread graph [9], $T_p^{\mathcal{P}}(\Pi, \chi, P)$ will be proportional to the computation demand of one critical path, $\Pi^k(P)$, in Π , that is

$$T_p^{\mathcal{M}}(\Pi, \chi, P) = C\Theta(\Pi^k(P), \chi).$$

Thus,

$$\mathcal{L}_a(\Pi, \chi, P) = C(\Theta(\Pi^k(P), \chi) - \frac{1}{P}\Theta(\Pi, \chi)).$$

For the same reason,

$$\mathcal{L}_a(\Pi, \chi', P') = C(\Theta(\Pi^k(P'), \chi') - \frac{1}{P'}\Theta(\Pi, \chi')).$$

Then we have

$$E_a(\Pi, \chi, P) = \frac{\frac{1}{P}\Theta(\Pi, \chi)}{\Theta(\Pi^k(P), \chi)},$$

and

$$Scale_a(E_a, (P, P')) = \frac{\Theta(\Pi^k(P), \chi) - \frac{1}{P}\Theta(\Pi, \chi)}{\Theta(\Pi^k(P'), \chi') - \frac{1}{P'}\Theta(\Pi, \chi')}.$$

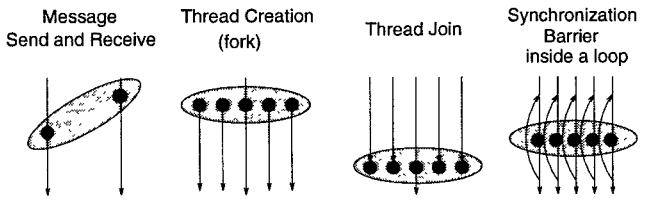

Figure 2: Use of thread graphs to model different communication patterns.

3 A Methodology for Scalability Prediction

The key to predict algorithmic and combination scalabilities is to effectively predict the parallel execution times of an application on the real machine, on the PRAM machine and on the MIRACLE machine. In this section, we first briefly introduce a semi-empirical methodology that we have used to predict the parallel execution times of applications on a real machine. Then we will introduce how this methodology can be adapted to predict parallel performances on the ideal machines.

3.1 Performance Prediction on Real Machines

In [9], a fairly complete model is developed to represent both explicit and implicit communications and synchronizations of a parallel program. This approach has achieved a good balance between using analytical techniques and experimental measurements in predicting multiprocessor performance. Our methodology is based on a two-level hierarchical model.

In the higher-level, a proposed graphical model called the *thread graph* is used to characterize parallel applications. In a thread graph, *threads* are used to model logical threads of control, *events* are used to model explicit communication and synchronization events in the threads of control, and *communication edges* are used to correlate relevant events and to express the cooperation among multiple threads of control. Figure 2 gives some examples of how thread graphs can be used to model different communication patterns. In Figure 2, a pointed line stands for a thread, a dark dot stands for an event, an oval stands for a communication, and a pointed arc stands for a loop. The examples in Figure 2, from left to right, depict how a communication edge and events are used to express separated message send and receive, thread fork, thread join, and synchronization barrier, respectively. Another key component of the higher-level model is a graph traversal algorithm. Once the elapsed times of all individual segments and events in the thread graph are estimated through the lower-level model, the graphical algorithm traverses a thread graph to estimate the parallel execution time of an application. At the same time, it takes into account the effects of various communication events and processor allocation strategies. The effects of the communication and synchronization events are handled according to their underlying semantics.

In the lower-level, the elapsed times of individual segments and events in the thread graph are determined with both analytic and experimental methods. In order to make our performance prediction methodology applicable to as many real architectures as possible, we construct the lower-level model according to the overhead patterns in different programming models. The basic idea to predict the elapsed times of the segments in a thread graph is based on the execution behavior of loop constructs. Because loops are repetitions of somewhat similar sequences of code blocks, in our study, we measure the elapsed times of the loop constructs with a small number of iterations to project the elapsed times of the loop constructs with a larger number of iterations. We refer to the measured data as empirical data. We obtain the empirical data by (1) modeling the computational complexity of all the segments in a thread graph in term of the application program parameters, χ, the number of processors, P, and the overhead patterns of the programming model used, (2) running a small number of instances of the problem on the real machine, and (3) solving the unknowns in the model by the measured elapsed times of the segments of the sample runs. Several factors will affect the accuracy of this method: (1) data-dependent computations inside a loop construct, so that different iterations of the loop are not identical, (2) dynamic system effects such as network latency, and (3) the effects of the memory hierarchy for memory-bound applications. According to our study, if the sizes of the sample runs are carefully chosen, the dynamic application and system effects can be partially captured in the empirical data.

Another important part of the lower-level performance model is the estimation of the elapsed times of all explicit communication events. At present, we use direct measurement to obtain the elapsed times of the explicit communication events involving different numbers of processors. We construct a dedicated execution environment for each type of communication event and measure its elapsed time directly. The reason for constructing such running environments is to eliminate the effects of load imbalance (for barrier) and contention (for lock), which have been partially taken care of by the graph traversal algorithm in the higher-level performance model.

3.2 Performance Prediction on Ideal Machines

To predict the parallel execution time of an application on the MIRACLE machine with P nodes, we first predict the elapsed time of an application on one node of the real machine, then divide the predicted elapsed time by the number of nodes of the MIRACLE machine, P. Because a MIRACLE machine will always exhibit linear speedup regardless of the characteristics of the program executed, the execution times of an application on one node of the MIRACLE machine and on one node of a real machine are the same.

To estimate the elapsed times of the segments on the PRAM machine that has no remote memory accesses, network latency and contention, we first serialize the thread graph and execute the program on one node of the real machine, and use the empirical data obtained on one node of the real machine for subsequent performance prediction. This will exclude the effects of any implicit communications. To eliminate the effects of network latency and contention, we assume the communication latency to be zero when using the graph algorithm to estimate the parallel execution times of the application on a PRAM machine.

4 Semi-Empirical Scalability Studies

In this section, we study the algorithmic scalability and the program-machine combination scalability of three applications implemented on two different machines. The algorithmic scalability evaluates the effects caused by the overhead patterns inherent in the program, which is independent of architecture. In practice, the algorithmic scalability can also

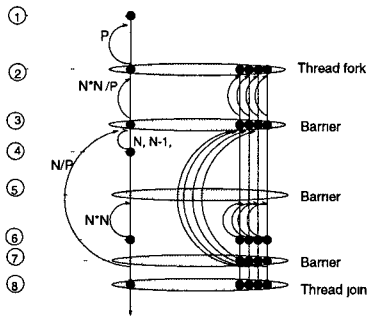

Figure 3: The thread graph representation of the GE program on the KSR-1.

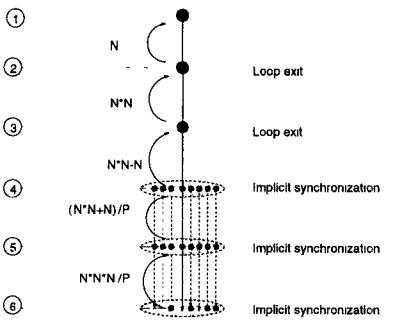

Figure 4: The thread graph representation of the GE program on the CM-5.

be used to examine the effects of programming models on algorithms. Because different programming model representations of an algorithm may present overhead patterns in different forms, the combination scalability reflects the effects caused by overhead patterns in both the program and the architecture. Combining both scalabilities, we can indirectly study the overhead patterns inherent in the architecture and its effects on the scalability of a program's execution.

The parallel architectures we used are the KSR-1 [3] that supports the shared-memory programming model and the CM-5 [8] that supports both message-passing and data-parallel programming models. The problems we used for scalability study are *All Pairs Shortest Path*, *Gauss Elimination* and a large *Electromagnetic Simulation* application [4].

4.1 Architectural Characteristics

The KSR-1 [3], introduced by Kendall Square Research, is a ring-based, cache coherent, shared-memory multiprocessor system with up to 1088 64-bit custom superscalar RISC processors(20MHz). A basic ring unit in the KSR-1 has 32 processors. The system uses a two-level hierarchy to interconnect 34 of these rings(1088 processors). Each processor has a 32 MByte cache and a 0.5 MByte subcache.

The CM-5 [8] is the newest member of the Thinking Machine's Connection Machine family. It is a distributed memory multiprocessor system which can be scaled up to 16K processors and supports both SIMD and MIMD programming models. Each CM-5 node consists of a SPARC processor operating at either 32 MHz or 40 MHz, 32 M-bytes of memory, and an interface to the control and data interconnection networks. The SPARC processor is augmented with four vector units, each with direct parallel access to the node's main memory. This yields an aggregated memory bandwidth of 256 MB/sec per processing node and a 128 MFLOPS peak floating-point rate per processing node.

4.2 Application Characteristics

The Gaussian Elimination (GE) algorithm mainly consists of an iteration of two parts: (1) determination of the pivot row and computation of the pivot column, and (2) elimination of the remaining columns by using the values in the pivot column. The shared-memory version of GE mainly parallelizes the elimination process by employing multiple threads, where multiple columns are eliminated simultaneously. In each iteration, all of the threads must wait at a

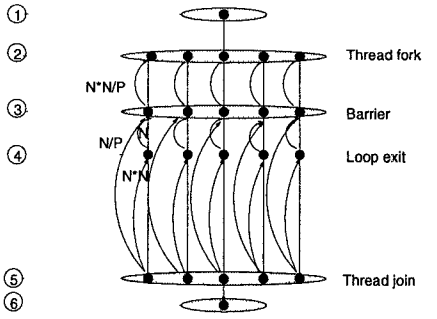

Figure 5: A thread graph representation of the APSP program on the KSR-1.

synchronization point for the main thread to do the pivot. To fully make use of the multiple threads, the initialization of the matrix is also parallelized. The thread graph representation of the shared-memory version of the GE program is shown in Figure 3, where, the thread in the left is the main thread, and the threads in the right form a thread cluster. P is the number of processors employed, and N is the size of the linear system. Edge $\langle 2, 3\rangle$ is the initialization, edge $\langle 3, 5\rangle$ is pivoting, and edge $\langle 5, 7\rangle$ is the elimination part. The expressions beside the pointed arcs are the estimated number of iterations of the loops in terms of P and N.

The data parallel version of GE is different from the shared-memory version. The initialization of the matrix is not parallelized. While the shared-memory version parallelizes the elimination of multiple columns, the data-parallel version parallelizes the elimination of the elements in each row(see Figure 4).

The All Pairs Shortest Path (APSP) problem calculates the shortest paths between all pairs of nodes in a weighted directed graph. The parallel algorithm of All Pairs Shortest Path is based on Dijkstra's sequential algorithm. Here we only implemented the shared-memory version of APSP on the KSR-1. It uses a path matrix to store the connections among nodes. For a given number of threads, the partition of the shortest path searching among threads can be done statically. Each thread is responsible for its shortest path. The program structure of the shared-memory version of the APSP program is as shown in Figure 5, which is mainly a thread cluster. In Figure 5, P is the number of processors employed and N is the size of the matrix.

The Electromagnetic (EM) simulation program simulates

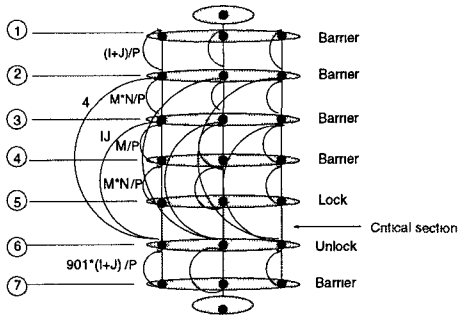

Figure 6: A thread graph representation of the EM simulation program structure on the KSR-1.

electromagnetic scattering from a conducting plane body [4]. In the simulation model, a plane wave from free space, defined as region "A" in the application, is incident on the conducting plane. The conducting plane contains two slots which are connected to a microwave network behind the plane. Connected by the microwave network, the electromagnetic fields in the two slots interact with each other, creating two equivalent magnetic current sources in the slots so that a new scattered EM field is formed above the slots. The shared-memory implementation of the EM program is mainly constructed by a thread cluster (see Figure 6). All edges except edge $\langle 5,6 \rangle$ are loops. Edge $\langle 5,6 \rangle$ corresponds to a critical section. At the end of each segment, there is a pair of events 'barrier-check-out' and 'barrier-check-in'. The input parameters of the EM simulation are I, J, M and N, which determine the problem size of the simulation.

For the data parallel version of the EM application, instead of employing multiple threads, large data sets are declared as parallel shaped data to be physically distributed across the processing nodes. Each processing node will perform operations simultaneously on its assigned section of the data sets. Repeated operations in different iterations in the sequential program are carried out concurrently by virtual processors in the system.

4.3 Validation of the Prediction Methodology

Here we give a brief summary of our validation results. In [10], we use simple metrics to study the effectiveness of the semi-empirical approach in predicting the parallel execution times. Our validation results indicate that:

- If the program abstraction is sufficiently detailed, the predicted results can be very precise. According to our experiments, the average error is less than 10% in most cases.
- The time reduction, which is defined as the ratio of the actual parallel execution time to the time to collect the sample data plus the time to estimate the parallel execution time, can be as high as hundreds for moderate problem sizes. It is expected to be much higher for larger problems.

Figures 7 - 9 compare the predicted and measured parallel execution times on the two machines when moderate number of processors were involved.

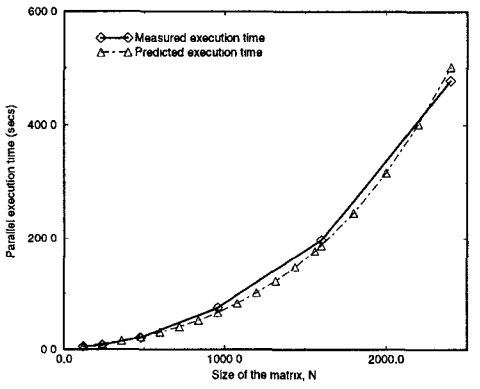

Figure 7: Predicted and measured parallel execution times of the GE program on the KSR-1 of 24 processors.

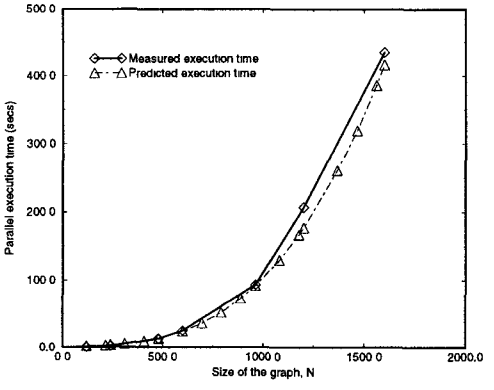

Figure 8: Predicted and measured parallel execution times of the APSP program on the KSR-1 of 24 processors.

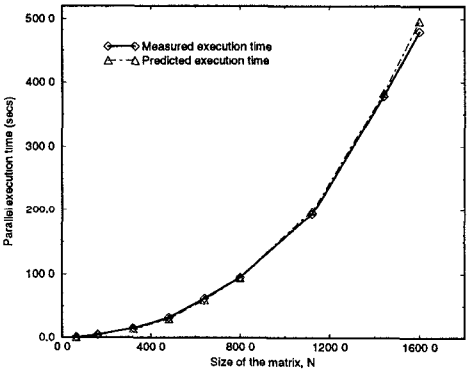

Figure 9: Predicted and measured parallel execution times of the GE program on the CM-5 of 128 processors.

P, P'	2	4	8	16	24	32	48
2	1.00	0.30	0.14	0.07	0.05	0.04	0.02
4		1.00	0.46	0.23	0.16	0.12	0.08
8			1.00	0.51	0.34	0.26	0.18
16				1.00	0.68	0.52	0.35
24					1.00	0.76	0.51
32						1.00	0.67
48							1.00
N	115	244	414	665	871	1055	1382

Table 1: The algorithmic scalability of the GE program with an algorithmic efficiency of 70% on the KSR-1.

P, P'	2	4	8	16	24	32	48
2	1.00	0.41	0.20	0.11	0.07	0.06	0.04
4		1.00	0.50	0.26	0.18	0.14	0.09
8			1.00	0.52	0.36	0.27	0.19
16				1.00	0.68	0.52	0.36
24					1.00	0.76	0.52
32						1.00	0.68
48							1.00
N	30	60	100	158	207	249	325

Table 2: The algorithmic scalability of the APSP program with an algorithmic efficiency of 70% on the KSR-1.

4.4 Performance Results on the KSR-1

The algorithmic scalabilities of the GE, APSP, and EM programs on the KSR-1 are summarized in Tables 1-3. The last row of each table gives the problem size (application parameters) needed to keep the efficiencies. In Table 3, the application parameters are given in the order I, J, M and N.

In Tables 1-3, the algorithmic efficiencies of all three programs are kept at a constant of 70%. Tables 1-3 indicate that all three programs are highly scalable in terms of their algorithm structures. Specifically, EM is the more scalable than APSP which is in turn more scalable than GE.

Tables 4-6 report the combination scalabilities of the shared memory implementations of the GE, APSP and EM programs on the KSR-1. In Tables 4-6, the parallel efficiencies for the three program machine combinations are all kept to 25%. (A higher efficiency than this was not possible for some programs.) Tables 4-6 show that the order of the combination scalabilities of the three programs on the KSR-1 are the same as the order of their algorithmic scalabilities. However, the difference among the combination scalabilities

P, P'	5	10	20	40	60
5	1.00	0.67	0.40	0.20	0.13
10		1.00	0.59	0.30	0.20
20			1.00	0.50	0.33
40				1.00	0.66
60					1.00
I, J	20, 20	20, 20	20, 20	20, 20	20, 20
M, N	1, 1	1, 1	80, 80	240, 240	384, 384

Table 3: The algorithmic scalability of the EM program with an algorithmic efficiency 70% on the KSR-1.

P, P'	2	4	8	16	24	32	48
2	1.0	0.14	0.02	0.01	0.001	0.0007	0.0001
4		1.00	0.14	0.02	0.007	0.0048	0.0007
8			1.00	0.16	0.051	0.0330	0.0050
16				1.00	0.313	0.2005	0.0305
24					1.000	0.6396	0.0974
32						1.0000	0.1522
48							1.0000
N	35	143	384	907	1539	1970	4238

Table 4: The combination scalability of the GE program on the KSR-1 with a parallel efficiency 25%.

P, P'	2	4	8	16	24	32	48
2	1.0	0.20	0.08	0.02	0.01	0.01	0.003
4		1.00	0.41	0.08	0.07	0.06	0.016
8			1.00	0.19	0.16	0.15	0.038
16				1.00	0.84	0.80	0.203
24					1.00	0.94	0.241
32						1.00	0.256
48							1.000
N	1	32	65	153	186	209	379

Table 5: The combination scalability of the APSP program on the KSR-1 with a parallel efficiency of 25%.

P, P'	5	10	20	40	60
5	1.00	0.85	0.53	0.178	0.075
10		1.00	0.62	0.209	0.088
20			1.00	0.335	0.142
40				1.000	0.422
60					1.000
I, J	20, 20	20, 20	20, 20	20, 20	20, 20
M, N	1, 1	1, 1	1, 1	16, 16	223, 223

Table 6: The combination scalability of the EM program on the KSR-1 with a parallel efficiency of 25%.

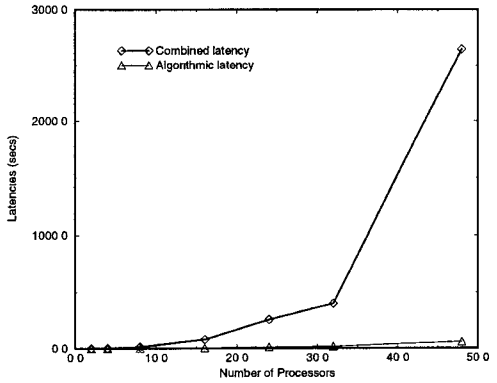

Figure 10: Algorithmic and parallel latencies of GE program on the KSR-1.

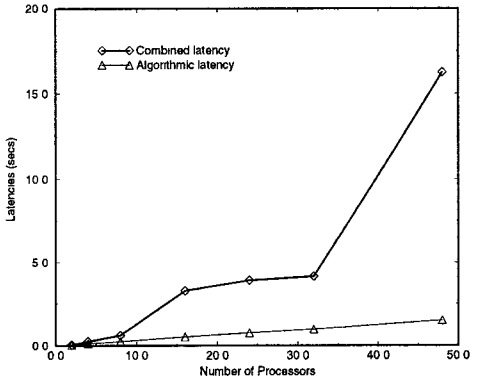

Figure 11: Algorithmic and parallel latencies of APSP program on the KSR-1.

on the KSR-1 is much larger than that of the algorithmic scalabilities. This indicates that a small difference in the synchronization and communication structures of a program can result in much larger difference in its performance on a real machine. In addition, we can see a significant decrease in scalability when the size of the machine is scaled from less than 32 nodes to more nodes. This can be further shown if we compare the algorithmic and parallel latencies when the parallel efficiencies are kept (see Figures 10-12). This is because remote memory access latency is higher when two rings are involved.

The gaps between the curves of algorithmic latency and parallel latency in Figures 10, 11 and 12 can be used to

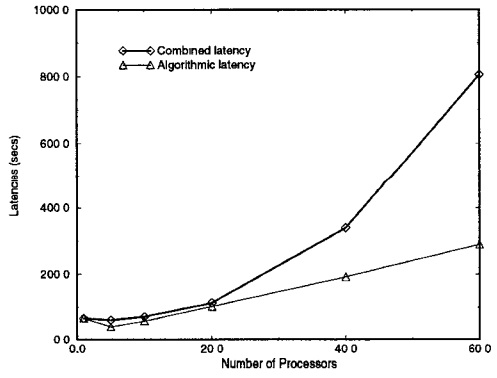

Figure 12: Algorithmic and parallel latencies of EM program on the KSR-1.

P, P'	32	64	128	512
32	1.000	0.235	0.057	0.0035
64		1.000	0.242	0.0148
128			1.000	0.0611
512				1.0000
N	14031	28717	58088	234317

Table 7: Algorithmic scalability of the data-parallel GE program with an algorithmic efficiency of 70% on the CM-5.

P, P'	32	64	128	512
32	1.000000	0.062220	0.004804	0.000286
64		1.000000	0.077216	0.004598
128			1.000000	0.059551
512				1.000000
N	125	488	1555	6509

Table 8: The combination scalability of the data-parallel GE program and the CM-5 with a parallel efficiency of 5%.

quantitatively compare effects of architecture on a program. The larger the gap, the more significant the overhead effects of the architecture will be. We will use this gap to make a further comparison of architecture effects on combination scalabilities between the KSR-1 and the CM-5.

4.5 Performance Results on the CM-5

We implemented a data-parallel version of the GE and EM programs on the CM-5. The algorithmic scalability of GE and its combination scalability are summarized in Tables 7 and 8. The two types of scalabilities for the EM program are reported in Tables 7 and 8.

The data-parallel version of the GE program exploited limited parallelism due to the nature of the computation. [9]. Therefore, both algorithmic scalability and the combination scalability are very low (see Tables 7-8). In contrast, the EM program is well suitable to the data-parallel model. Thus, both scalabilities are high (see Tables 9 and 10).

Figures 13 and 14 compare the algorithmic and combination latencies of the GE and EM programs on the CM-5. Figure 13 shows a significant difference between algorithmic and parallel latencies. This is because the only parallel parts of the data-parallel GE program are the swaps in pivot and the simultaneous elimination of the elements of each column. The parallel part is a small portion of the execution of the program. This resulted in a large percentage of synchro-

P, P'	32	64	128	256	512
32	1.00	0.2065	0.1148	0.0939	0.086
64		1.0000	0.5559	0.4549	0.416
128			1.0000	0.8183	0.749
256				1.0000	0.915
512					1.0000
$I = J$	1	1371	5770	14569	32226
$M = N$	20	20	20	20	20

Table 9: Algorithmic scalability of the data-parallel EM program with an algorithmic efficiency of 90% on the CM-5.

P, P'	32	64	128	256	512
32	1.00	0.19	0.11	0.087	0.080
64		1.00	0.55	0.448	0.410
128			1.00	0.816	0.746
256				1.000	0.914
512					1.000
$I = J$	1	1371	5770	14569	32226
$M = N$	20	20	20	20	20

Table 10: Combined scalability of the data-parallel EM program and the CM-5 with a parallel efficiency 90%.

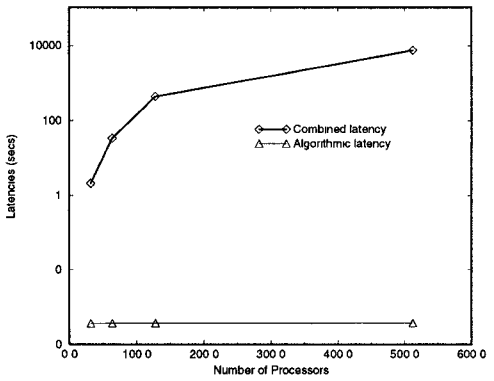

Figure 13: Algorithmic and parallel latencies of GE program on the CM-5.

nization overhead. The larger the application parameter, the larger the number of iterations, and hence the larger the synchronization overhead.

In contrast, Figure 14 shows a very small difference between algorithmic and parallel latencies in the EM program on the CM-5. Both latencies are much smaller than the GE program on the CM-5. This is because the parallelized portion of the EM program is much larger than that of the data-parallel GE. (Almost all the computation that is parallelizable using the shared-memory model is parallelizable in the data-parallel version.) The other two factors that contribute to the high combination scalability are the small percentage of synchronization overhead of the program and the highly scalable Fat-tree networks of the CM-5 [9].

In this section, because we can not obtain empirical data on a single node of the CM-5. the empirical data on 32 nodes CM-5 were used instead to approximate the execution times

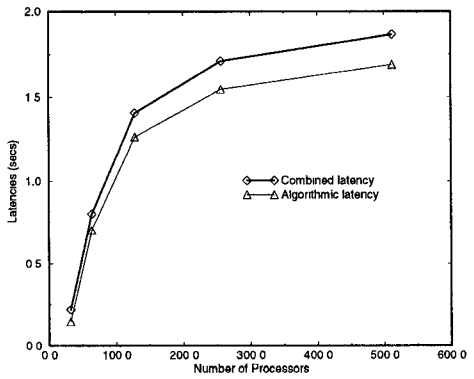

Figure 14: Algorithmic and parallel latencies of EM program on the CM-5.

program	KSR-1 GE	KSR-1 APSP	KSR-1 EM	CM-5 GE	CM-5 EM
experiment	396	53653	2494	60	8256
prediction	1.1	0.76	0.1	1.0	8.0

Table 11: Time comparisons between using the semi-empirical prediction method and using pure experimental method to obtain the combination scalabilities of the three programs on the two machines (in secs).

on one node of the ideal machines. Thus the performance result in this section could be more optimistic compared with that on the KSR-1.

4.6 Time Reduction of Scalability Prediction

To show the time reductions of the semi-empirical approach in predicting the combination scalabilities of the GE, APSP and EM programs on the KSR-1 and CM-5, Table 11 compares the times to obtain the scalabilities by prediction with the times to obtain the scalabilities by experiments. In Table 11, we assume that the proper problem size can be decided at once, which is the best case for the experimental method. In Table 11, the time to measure the empirical data is also excluded from the prediction times.

4.7 Comparisons

Using different program implementations on the two different machines, we are able to compare the effects of the shared-memory and data-parallel programming models on a program's algorithmic scalability, and the effects of the KSR-1 and the CM-5 on a program's combination scalability. Here, we summarize our comparisons:

- Different programming models can significantly affect the algorithmic scalability of a program because the parallelism inherent in the program can be exploited differently. For instance, to keep the algorithmic efficiency up to 70%, the algorithmic scalability of the data-parallel GE program was 0.234951 when the machine size scales from a small to a moderate number of processors. The scalability was 0.003481 when the machine size scales from a small to a large number of processors (see Table 7). In contrast, the algorithmic scalability of the shared-memory GE program was 0.45 when the machine size scales from a small to a moderate number of processors. The scalability was 0.08 when the machine size scales from a small to a large number of processors (see Table 1). We can make another comparison between between Tables 1 and 7. For 32 processors, to keep the same algorithmic efficiency, the problem size is 1055 for the shared-memory implementation, but it is 14031 for the data-parallel implementation — about 14 times larger. This indicates that the data-parallel GE program is significantly less scalable than that of the shared-memory version.

- Even though the differences among algorithmic scalabilities can be moderate for the different implementations of a program, the combination scalabilities can be significantly different (see Tables 1, 7, 4 and 8). This indicates that different communication and synchronization pattern and structure, and also different

architectural support, can result in significantly different performance. By comparing the latency curves and by analyzing the communication pattern and structure of the program, we can easily locate any performance bottlenecks existing in different program implementations.

- Based on a careful examination of the scalability tables, the latency curves, and the computation and communication structure of the programs, we conclude that, (1) the EM application is more suitable to be implemented in the data-parallel model because of its large number of parallelizable data sets, and its larger grain of parallelizable computation. (2) The communication structure of the CM-5 is highly efficient due to the Fat-tree data network and the support of a control network. (Very small differences between algorithmic and parallel latencies are shown for the EM program we ran on the CM-5.) (3) For applications with a small parallelizable data set, the synchronization overhead can become a bottleneck if the grain of parallel computation is small, as in, for example, the data parallel GE program on the CM-5. (4) The combination scalability of a program is highly architecture dependent. We have shown performance differences of a program on the two different architectures.

5 Conclusion

Effective use and refinement of scalability metrics and evaluation methods is a major issue of parallel performance evaluation. To address limitations in the scalability study, we have extended the latency metric in [11] by providing a more precise definition of problem size, and by quantifying algorithmic scalability based on algorithmic latency and efficiency. The algorithmic scalability defined in this paper is quite independent of the hardware architecture. To effectively estimate the various scalabilities, a semi-empirical approach is used. The semi-empirical approach can be used to estimate the various scalabilities on systems with implicit communications. The time to estimate the scalabilities is significantly reduced compared with pure experimental and simulation methods. For instance, it took 395.5 seconds to obtain the combination scalability of a shared memory implementation of the GE program on the KSR-1, and 53652.7 seconds for the combination scalability of the APSP on the KSR-1 using the pure experimental method in the best case. In contrast, it only took 1.08 and 0.76 seconds respectively using the semi-empirical method (the time to collect the sample data is excluded) — approximately 366 times faster for the GE program and 70595 times faster for the APSP program. The representation of a program as a thread graph enables the program overhead pattern of the program to be studied according to its synchronization and communication structure. Because experimental measurement is used to estimate the complex system effects, this makes our architecture model less dependent on the real architectures. Thus, it is more flexible for the evaluation of a large variety of hardware architectures. Because important and implicit system effects are obtained through experimental measurements, the semi-empirical performance model is more precise than pure analytical models or simulations. There are still some limitations of the semi-empirical approach. First, the communication structure of the program should be as regular as possible. Second, the number of iterations of the major loop in terms of application parameters and system size should be easily determined by static analysis, or by program slicing with small overhead. Finally, changing application parameters should not significantly change the communication pattern of the program.

References

[1] A. Grama, A. Gupta and V. Kumar, "Isoefficiency function: a scalability metric for parallel algorithms and architectures", *IEEE Parallel & Distributed Technology*, Vol. 1, No. 3, 1993, pp. 12-21.

[2] J. L. Gustafson, "The consequences of fixed time performance measurement", *Proceedings of the 25th Hawaii International Conference on System Sciences*. Vol. III, 1992, pp. 113-124.

[3] Kendall Square Research, KSR-1 *Technology background*, 1992.

[4] Y. Lu, et. al., "Implementation of electromagnetic scattering from conductors containing loaded slots on the Connection Machine CM-2", *Proceeding of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, SIAM Press, 1993, pp. 216-220.

[5] D. Nussbaum, A. Agarwal, "Scalability of parallel machines", *Communication of the ACM*, March 1991, Vol. 34, No. 3, pp. 57-61.

[6] X. Sun and D. T. Rover, "Scalability of parallel algorithm-machine combinations", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 6, 1994, pp.599-613.

[7] A. Sivasubramaniam, A. Singla, U. Ramachandran, H. Venkateswaran, "An approach to scalability study of shared memory parallel systems", *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1994, pp. 171-179.

[8] Thinking Machine Corporation, *The Connection Machine CM-5 Technical Summary*, 1993.

[9] X. Zhang, Z. Xu, L. Sun, "Performance prediction on implicit communication systems", in *Proceedings of the Sixth IEEE Symposium of Parallel and Distributed Processing*, IEEE Computer Society Press, Oct. 1994, pp. 560-568.

[10] X. Zhang, Z. Xu, L. Sun, "Semi-empirical multiprocessor performance predictions", Technical Report, High Performance Computing and Software laboratory, The University of Texas at San Antonio, March 1995.

[11] X. Zhang, Y. Yan, K. He, "Latency metric: an experimental method for measuring and evaluating parallel program and architecture scalability", *Journal of Parallel and Distributed Computing*, Vol. 22, No. 3, 1994, pp. 392-410.

[12] X. Zhang, Y. Yan and K. He, "Evaluation and measurement of multiprocessor latency patterns", in *Proceedings of the 8th International Parallel Processing Symposium*, IEEE Computer Society Press, April, 1994, pp. 845-852.