JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **39,** 14–28 (1996)
ARTICLE NO. 0151

Semi-empirical Multiprocessor Performance Predictions[1]

ZHICHEN XU,* XIAODONG ZHANG,† AND LIN SUN‡

**Computer Sciences Department, University of Wisconsin—Madison, Madison, Wisconsin 53706; †High Performance Computing and Software Laboratory, The University of Texas at San Antonio, San Antonio, Texas 78249; and ‡Cap Gemini America, Denver, Colorado 80503*

This paper presents a multiprocessor performance prediction methodology supported by experimental measurements, which predicts the execution time of large application programs on large parallel architectures based on a small set of sample data. We propose a graph model to describe application program behavior. In order to precisely abstract an architecture model for the prediction, important and implicit architecture parameters are obtained by experiments. We focus on performance predictions of application programs in shared-memory and data-parallel architectures. Real world applications are implemented using the shared-memory model on the KSR-1 and using the data-parallel model on the CM-5 for performance measurements and prediction validation. We show that experimental measurements provide strong support for performance predictions on multiprocessors with implicit communications and complex memory systems, such as shared-memory and data-parallel systems, while analytical techniques partially applied in the prediction significantly reduce computer simulation and measurement time. © 1996 Academic Press, Inc.

1. INTRODUCTION

1.1. Background

Two classes of methods have been used to predict multiprocessor performance: methods that employ simulation tools to construct and run a detailed system model and methods that use analytical techniques [6]. Simulation is quite expensive in terms of its consumption of computing resources, and it may be more suitable to focus on studying a certain part of architecture and system performance. Analytical methods that use a single model to abstract both the application and the architecture usually construct a single overhead function to capture the overall overheads inherent in both parts. Such a single model is often too simple to describe practically the complexity of application programs and parallel architectures [4].

To have a better control over complex application and architecture behavior, most analytical methods use hierarchical models. In [3], Adve provides a framework for parallel program performance prediction models which well characterizes most of the existing models by a hierarchy of higher and lower level models. In the higher-level component, task graphs [11, 13] are usually used to represent the task-level behavior of the program. A task graph is a directed acyclic graph in which each vertex represents a task and each edge represents the precedence relationship between a pair of tasks. There is no internal parallelism inside a task, and a task must be executed sequentially. This higher-level model component computes the overall execution time assuming individual task execution times are known. The lower-level components represent system-level effects and are usually simulated by stochastic processes [6, 13] or by some type of system overhead function [10]. Individual task execution times are computed from lower-level components.

Thomasian and Bay [13] propose a two-level model for a class of programs which can be represented by directed acyclic graphs. At the higher level, the system behavior is specified by a Markov chain whose states correspond to the combination of tasks in execution. At the lower level, the transition rates among the states of the Markov chain are computed using a queueing network solver, which determines the throughput of the computer system for each system state.

Vrsalovic *et al.* [15] develop an analytic model for predicting the performance of iterative algorithms. Using the same approach, we predict the execution performance of a program with a larger number of iterations based on the performance of the same program with a small number of iterations. However, the method in [13] focuses on iterative algorithms and models the decomposition of a program into processes by using pure analytic functions. The model proposed in this paper focuses on broader categories of applications. The hierarchical model in this paper distinguishes deterministic factors from non-deterministic performance factors, and implicit communications from explicit communications. Both analytic and experimental methods are combined performance prediction.

Kapelnikov *et al.* [6] propose a methodology that embodies two modeling domains: the *program domain* and the *physical domain*. In the program domain, a graphical model

[1] This work is supported in part by the National Science Foundation under Grants CCR-9102854 and CCR-9400719, by the Air Force Office of Scientific and Research under Grant AFOSR-95-1-0215, 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.

called a *computation control graph* is used to model a program's structure. Some novel features of computation control graphs include flexibility in modeling loop constructs and multiple instantiations of tasks. The physical domain consists of a queueing network that represents both system resources and synchronization constraints that are related to tasks' interdependencies. Performance prediction involves the process of constructing a Markov process whose state space consists of all relevant states of program execution, solving the closed queueing network representation of the physical domain model to obtain system throughputs, approximating state transition rates from the system throughputs, and solving the Markov process to obtain an estimate of the average program execution time.

Tsuei and Vernon [14] propose a hierarchical analytical model to evaluate the *relative importance* of factors that can limit speedup on MIMD shared memory multiprocessors. Specifically, they use the hierarchical model to estimate the relative impact of software structure, lock contention, and hardware resource contention on speedup. The effect of each factor in reducing the speedup is expressed as a multiplicative *efficiency factor.* For a fairly general class of parallel programs, the hierarchical model can be solved without iterating among the submodels.

Adve [3] develops a simple deterministic model for parallel program performance prediction. A task-graph-based representation is used to represent both program parallelism and scheduling. A graph solution algorithm is used to estimate the parallel execution time of a program. Deterministic values represent mean task times, including communication, and shared-resource contention computed from a separate and stochastic model. The model is used to predict the impact of system changes as well as program design changes that affect load-balancing. In this paper we propose a similar graph traversal algorithm for our thread graph representation of a program. Instead of the system-level model being solved on the fly, most of the non-deterministic factors are taken care of through a lower-level model before the graph traversal begins.

Menasc̃e *et al.* [10] also use a task graph for their higher-level model component. Instead of using the Markov chain and queueing network, Menasc̃e's methodology estimates the execution time of the parallel program based on the interdependency between network delay and program execution. In [10], network delay is modeled as a function of the network message injection rate, which in turn depends on the total communication demand and estimated execution time of the program. The estimated execution time is affected by the network delay. An iterative method is used to solve a fixed-point system of equations.

There are several limitations to pure analytical methods in general. First, analytical methods that use queueing models make many assumptions about the application and architecture. These assumptions may bring inaccuracies to the prediction results. Second, methodologies that depend on precise system overhead functions in terms of the computation and communication demands of the applications would be difficult to apply widely, because it is not easy to extract the communication demand of a parallel program accurately. For example, in shared-memory machines such as the KSR-1, a large amount of implicit network traffic and contention can come from both explicit synchronization statements and implicit communication effects such as cache coherence. These implicit network activities are difficult to capture.

1.2. Semi-empirical Methodology

Instead of solely using analytical techniques, in our performance prediction approach analytical methods are used to capture as many deterministic performance factors as possible. Implicit and ''non-deterministic'' application and architecture parameters are obtained through experiments. Analytical and empirical results are combined to predict performance.

Like most of the previous work, our methodology is also based on a two-level hierarchical model. In the higher level, a graphical model called the *thread graph* is proposed and is used to characterize parallel applications, and a graphical algorithm is used to estimate the parallel execution time of a parallel application, assuming the elapsed times of all individual segments and events in the thread graph are known. Some novel features of the thread graph include (i) the explicit representation of explicit communication and synchronization events, which enables all such communications and synchronizations to be captured and handled differently according to their underlying semantics in determining the execution time of the application, and (ii) the incorporation of a *thread class* and a *cluster* that enable our graphical approach to model various language constructs. The graphical algorithm traverses a thread graph to estimate the parallel execution time of an application, while at the same time it takes into account the effects of various communication events and processor allocation strategies. On the lower level, the elapsed times of individual segments and events in the thread graph are determined with both analytic and experimental methods. Implicit and non-deterministic system effects are obtained through experimental measurements. With our performance prediction methodology, the performance of large applications on parallel architectures can be predicted based on a small amount of sampling data.

The rest of the paper is organized as follows. Section 2 gives a detailed description of our performance model and methodology. Section 3 briefly describes the test beds for the prediction model, the KSR-1 and the Connection Machine CM-5, and introduces the test seeds, Gauss elimination (GE), all pairs shortest path (APSP), and electromagnetic scattering simulation application (EM) programs. Section 4 presents the validation and performance prediction results. Section 5 is a discussion of the difficulties and limitations of the semi-empirical approach. Section 6 summarizes the work.

2. THE PERFORMANCE MODEL AND METHODOLOGY

Many factors affect the performance of an application on a parallel architecture. The major ones are (1) the computation and communication (both explicit and implicit) demands imposed on a parallel architecture by an application and (2) how the parallel architecture is able to fulfill these demands. In this section, we attempt to develop a fairly complete representation of computation and communication demands of a parallel application and to achieve a good balance between analysis and measurement to estimate the ability of an architecture to fulfill these demands.

2.1. The Higher-Level Performance Model

Because most variations of the task graph have neglected the detailed semantics of most communication and synchronization events, a novel program representation called the *thread graph* is proposed. The thread graph provides more accurate and detailed information about the communication and synchronization events of an application. The thread graph can be seen as a ''transformation'' or ''explanation'' of a task graph.

2.1.1. Thread Graph. In the thread graph, a *thread* is an abstraction of a logical thread of control. This concept is slightly different from a thread in an operating system, which is a more physical concept. For example, a thread in Unix is a lightweight process that consists of a program counter, private data and a stack. *Events* are used to represent communication points or control transitions in a thread of control. Examples of events that mark control transitions include entry and exit points of a loop and the starting and ending points of an application. *Communication edges* consisting of communication events are used to correlate relevant events and model cooperations among the multiple threads of control. With the thread graph, different types of communication events in an application can be classified and treated differently. Figure 1 gives some examples of how thread graphs can be used to model different methods of communication. In Fig. 1, 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 figures in Fig. 1, from left to right, depict how a communication edge and events are used to express separated message send and receive, thread creation (fork), thread join, and synchronization barrier, respectively.

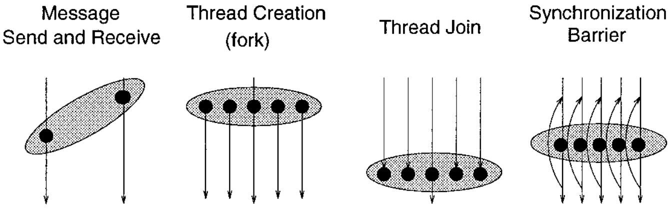

FIG. 1. Use of thread graphs to model different communication patterns.

In general, the number of threads of control involved in a computation is not fixed. In a thread graph, the thread *class* and *cluster* are used to model those language mechanisms that can adapt to system size. An example of such language mechanisms is a *parallel region* of KSR Fortran [1]. A thread class is a template that can be instantiated into different numbers of threads. A thread cluster is instantiated from a thread class and dedicated to a common goal. We will refer to the template that represents a class of threads as a *class thread,* and a thread that is instantiated from a class thread as an *instance thread*. Similarly, an event or a segment in a class thread is called a class event or segment, while an event or a segment in a instance thread is called an instance event or segment.

Since loops that contain explicit communication events can significantly affect the execution behavior of a parallel program, such loops are expressed explicitly in the thread graph. Loop entry and loop exit are special events that represent control transitions. Both the entry and the estimated number of iterations of a loop are recorded as attributes in the corresponding loop exit event. Another attribute of a loop exit event is the remaining number of iterations of the corresponding loop. It is included to facilitate the traversal of a thread graph and is initialized to be the estimated number of iterations of the loop.

Formally, a thread graph is a 3-tuple, $\Pi = (\Gamma, \Phi, \bowtie)$, where

- $\Gamma = \{\Gamma_i\}$ is an ensemble of parallel threads that cooperate to solve the problem. Each Γ_i is in turn separated by multiple *events* into segments $\{s_i^j\}$, $i = 1, 2, ..., n$, where n is the total number of threads, and $j = 1, 2 ..., k_i$, where k_i is the total number of segments of the thread Γ_i. In order that the structure of a program can be represented in a hierarchical way, a segment in a thread graph can itself be a thread graph or a thread cluster. This also provides a way to express recursiveness. A thread graph is said to be *fully instantiated* if all segments in it have no internal parallelism. In the rest of the paper, when we talk about a thread graph, we will assume it is a fully instantiated thread graph unless otherwise stated. When a segment corresponds to a thread cluster, the symbol $\hat{s}$ is used instead. The symbol $\hat{\Gamma}$ will denote a thread class or cluster. Sometimes a segment can also be expressed as an ordered pair of events, such as $\langle e_s, e_t \rangle$, where e_s and e_t are events on the same thread.

Let S represent all segments in the parallel program Π.

- Φ is a set of *events.* An *event* either marks explicit communication or transition of control. The events that mark control transition include the starting and ending points of an application, entry and exit points of a loop, and branch points of code blocks.
- $\bowtie = \{c_i\}$ is a set of communication edges that denotes explicit cooperation among the parallel threads, where

each c_i is in the form $\langle e^{j_1}_{i_1}, e^{j_2}_{i_2}, ..., e^{j_r}_{i_r}\rangle$, where in turn $e^{j_k}_{i_k}$ is an event on the thread Γ_{i_k}, $1 \leq k \leq r$, and r is the number of threads that are correlated by the communication edge c_i. In general, $\Gamma_{i_j} \neq \Gamma_{i_l}$ when $j \neq l$.

2.1.2. Estimate of Parallel Execution Time. Once an application program is represented by a thread graph Π, the following steps are conducted to obtain the parallel execution time of the application with input parameter χ and the number of processors N:

1. Fully instantiate the thread graph Π according to the input χ and number of processors N.
2. Obtain the elapsed times of all segments and events, denoted by $\text{tt}(\Pi, \chi, N)$ and $\text{tt}^{\text{e}}(\Pi, \chi, N)$ through the lower-level model.
3. Use a graphical algorithm, **G**, to traverse the fully instantiated thread graph to estimate the parallel execution time of the application. The inputs of **G** are the fully instantiated thread graph, $\Pi(\chi, N)$, the elapsed times of all segments, $\text{tt}(\Pi, \chi, N)$, the elapsed times of all events, $\text{tt}^{\text{e}}(\Pi, \chi, N)$, and the processor scheduling strategy. When several ready events occurs at the same time or there are more threads than the number of processors, the processor scheduling strategy takes this into effect. The scheduling policies are incorporated into algorithm **G**. So far we have assumed a *random policy* in selecting the ready events, and assumed a *time-sharing policy* to deal with the situation that the number of executable threads is larger than the number of processors available. That is, all ready events that occur at the same time have the same probability to be selected as the next events to be processed, and all active threads share the available processor resources with the same priority when there are more active threads than available processors. Thread migration is not considered in algorithm **G**. It is possible to further incorporate new scheduling strategies with minor changes to algorithm **G**. Table I is a description of algorithm **G** at a high-level.

To obtain the time complexity of algorithm **G**, for any event $e \in \Phi$, we define $L(e)$ as a set of loops that encompass event e in a thread graph. $L(e)$ is empty if e is outside any loop construct. For any loop l in a thread graph, we define $\text{Size}(l)$ as the number of iterations of loop l. Then, the number of times an event e is processed in traversing the thread graph when carrying out the algorithm **G**, $\theta(e)$, will be

$$\theta(e) = \prod_{l_i \in L(e)} \text{Size}(l_i). \tag{2.1}$$

Thus, the total number of events that are encountered when traversing a thread graph, $\Theta(e)$, is

$$\Theta(e) = \sum_{e \in \Phi} \theta(e) = \sum_{e \in \Phi} \left(\prod_{l_i \in L(e)} \text{Size}(l_i) \right). \tag{2.2}$$

TABLE I
A Description of G at Higher Level

```
1: {Put all ready events into a ready event pool.}
2: While (there are ready events) do
3: {
4:    Pick up one ready event, say $e_s$, according to the scheduling policy.
5:    Process the event, $e_s$, according to its underlying semantics.
6:    /* Check for more ready events, and put them into the ready event pool.*/
7:    Let $e_t$ be the event that immediately follows $e_s$ in the thread.
8:    {
9:       If ($e_s$ is not a loop exit, or the remaining number of iterations of the loop is zero)
10:      {
11:            Put $e_t$ into the ready event pool.
12:            Reset the remaining number of iterations of the loop, if $e_s$ is a loop exit.
13:      }
14:      If ($e_s$ is a loop exit, and the remaining number of iterations of the loop is nonzero)
15:      {
16:            Decrement the remaining number of iterations of the loop by one.
17:            Put the corresponding loop entry event into the ready event pool.
18:      }
19:   }
20: }
```

The time complexity of **G** is $\Theta(e)$. $\Theta(e)$ is a function of the application parameters χ and the number of available processors N. The algorithm **G** can be seen as a kind of measurement-based simulator, so in the worst case, $\Theta(e)$ will be of the same order as the time complexity of the application being modeled. Fortunately, in most cases what we need is only a very high-level abstraction of the program, and the execution behavior of a thread cluster can be approximated with that of one representative thread in the cluster. This will greatly reduce the number of events that must be traversed. One thing that worth mentioning here is the time complexity of carrying out steps 4 and 5 in the **G** algorithm. In most cases, it is a constant or proportional to the number of processors involved. And in many cases, it can be made into a constant by sacrificing extra space. For example, we can use hash tables for messages and global data structures for barriers or other global communication events.

2.2. The Lower-Level Performance Model

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. In the lower-level performance model both analytic and measurement methods are combined to capture the ability of a parallel architecture to fulfill the computation and communication demands imposed by an application.

2.2.1. Shared-Memory Programming Model. In a shared-memory machine, many implicit communication activities will affect the performance of an application. These include remote memory accesses and cache coherence activities.

The estimation of the elapsed time of segments in a thread graph is based on the execution behavior of loop constructs, which are the repetition of the *same*[2] code blocks. A segment, denoted by s, in a fully instantiated thread graph can be separated into subsegments of sequential code blocks and loop constructs. Because a loop is, to a great extent, the repetition of the same code blocks, the basic idea of our lower-level performance model is to use the performance of a small number of iterations of a loop to estimate the performance of the loop with a larger number of iterations. Each iteration of a loop construct in a shared-memory program basically consists of two major parts: local processing and remote memory accesses. The random effects inside a loop include the dynamic application effects introduced by the conditional statements and system random effects introduced by remote memory accesses. By measuring the performance of a loop with a small number of iterations, we hope these empirical data will imply both the deterministic and random effects and can be used to estimate the performance of the loop with a larger number of iterations.

A given segment, s, in a fully instantiated thread graph can be separated into a set of subsegments, denoted by $\{ss\}$. Each ss is either a sequential code block or a loop construct. If ss is a sequential code block, we can regard it as a loop construct that has a constant number of iterations, namely just 1. For each $ss \in s$, we define $\mathscr{F}(ss, \chi, N)$ to be the estimated number of iterations that ss executes when the application parameters are χ and the number of available processors is N. Let the symbol $\mathrm{tt}(ss, \chi, N)$ denote the elapsed time of ss with the application parameter χ on N processors. Then $\mathrm{tt}(ss, \chi, N)$ can be approximated with the formula

$$\mathrm{tt}(ss, \chi, N) \doteq \frac{\mathscr{F}(ss, \chi, N)}{\mathscr{F}(ss, \chi_0, N_0)} \mathrm{tt}(ss, \chi_0, N_0), \tag{2.3}$$

where χ_0 is a set of application parameters with values smaller than those of χ, and N_0 is the number of processors involved in measuring the empirical data.[3]

Once the elapsed time of all subsegments of segment s is estimated, the elapsed time of segment s with application parameters χ on N processors, denoted by $\mathrm{tt}(s, \chi, N)$, can be estimated with the formula

$$\mathrm{tt}(s, \chi, N) = \sum_{ssi \in s} \mathrm{tt}(ss, \chi, N). \tag{2.4}$$

In practice, to obtain $\mathrm{tt}(s, \chi, N)$, we can group the subsegments that have the same $\mathscr{F}(ss, \chi, N)$ together and model $\mathrm{tt}(s, \chi, N)$ as a function of χ, N and some undecided empirical parameters—in other words, analyze the computational complexity of the entire segment in terms of χ and N. Instead of measuring the elapsed times of all individual subsegments, we can measure the elapsed times of the entire segment s with a small set of application parameters of small value and solve the undecided empirical parameters. An interesting problem motivated by this is how to extract and apply application-independent empirical data, greatly reducing prediction time.

When a thread cluster $\hat{\Gamma}$ executes without explicit communication with other threads, we can determine the elapsed time of the cluster as a whole. To do this, we can approximate the elapsed time of the cluster with the elapsed time of *a representative thread* in the cluster. We usually choose a thread in a cluster that has the maximum work load as the representative thread. It can also be chosen according to different situations. Special care must be taken to treat segments that cannot execute concurrently with other threads (segments corresponding to *critical sections* in the shared-memory programming model) and to treat effects of the number of events in the communication edges. The elapsed time of $\hat{\Gamma}$ can be approximated as

$$\begin{aligned} \mathrm{tt}(\hat{\Gamma}, \chi, k) \approx & \sum_{s \in S^p(\Gamma)} \mathrm{tt}(s, \chi, k) \\ & + k \sum_{s \in S^s(\Gamma)} \mathrm{tt}(ss, \chi, k) \\ & + \sum_{e \in E(\Gamma)} \mathrm{tt}(e, \chi, k), \end{aligned} \tag{2.5}$$

where Γ is a representative thread in $\hat{\Gamma}$, k is the number of instance threads in the cluster, $\hat{\Gamma}$, $S^s(\Gamma)$ is the set of segments in Γ that cannot be executed concurrently, $S^p(\Gamma)$ is the set of segments in Γ that execute concurrently with other threads, and $E(\Gamma)$ is the set of events in Γ.

Another important part of the lower-level performance model is the estimation of the elapsed times of all explicit communication events. In this paper, 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 taken care of by the graph algorithm **G**.

[2] Different iterations may not be exactly the same, due to conditionals and dynamic system effects.

[3] Note that the choice of χ_0 is very important for a good estimation. Several factors that will affect the precision of the prediction include dynamic application factors, such as data-dependent computation, and dynamic system effects, such as the effect of the memory hierarchy (see discussion).

2.2.2. Data-Parallel Programming Model. The major overheads in the execution of a data-parallel program include implicit synchronizations and communications. Segment s can be separated into a set of subsegments, $\{ss\}$, where each ss is either a sequential code block or a parallel code block. Similar to the shared-memory programming model, a sequential code block can be regarded as a special parallel code block with parallelism of one. A parallel code block is similar to a thread cluster in a shared-memory program, but simpler. A thread cluster here is simply a loop construct that is distributed among multiple processors. We define $\mathscr{F}(ss, \chi)$, the *size* of a parallel code block ss, as the estimated number of iterations of the corresponding loop construct. When there are N processors involved in executing the subsegment, the number of iterations one processor will carry out is $\lceil \mathscr{F}(ss, \chi)/\min(\mathscr{F}(ss, \chi), N) \rceil$. Thus, the elapsed time of ss with application parameter χ and machine size N, $\mathrm{tt}(ss, \chi, N)$, can be approximated with the formula

$$\mathrm{tt}(ss, \chi, N) = \left\lceil \frac{\mathscr{F}(ss, \chi)}{\mathscr{F}(ss, \chi_0)} \frac{\min(N_0, \mathscr{F}(ss, \chi_0))}{\min(N, \mathscr{F}(ss, \chi))} \right\rceil \mathrm{tt}(ss, \chi_0, N_0), \tag{2.6}$$

where χ_0 is a set of application parameters of values smaller than the ones of χ, and N_0 is the machine size used to measure the empirical data. The effects of explicit communications and dynamic system effects are handled in the same way as in the previous section. Once the elapsed times of all the subsegments of a segment have been estimated, the elapsed time of the segment s itself is then approximated by

$$\mathrm{tt}(s, \chi, N) = \sum_{ss \in s} \mathrm{tt}(ss, \chi, N) + O_{\mathrm{syn}}, \tag{2.7}$$

where O_{syn} is the accumulation of synchronization overheads, vector startup times, and instruction dependency overheads. This can be obtained by direct measurement or solved as an empirical parameter. We notice that a different segment may have a different value of O_{syn}. Similar to the lower-level performance model for shared-memory programs, we can group the subsegments that have the same $\mathscr{F}$ together, model the $\mathrm{tt}(s, \chi, N)$ as a function of χ, N and some undecided empirical parameters, and solve the unknowns through a small number of small-size sample runs.

3. TEST BEDS AND SEEDS OF PERFORMANCE PREDICTION METHODOLOGY

We validated the semi-empirical performance prediction methodology by comparing the measured and predicted parallel execution times of three different applications on two different parallel machines. The parallel architectures we used as the test beds are the KSR-1 [1], which supports shared-memory programming model, and the CM-5 [2], which supports both message-passing and data-parallel programming models. The problems we used as test seeds are *Gauss elimination* (GE), *all pairs shortest path* (APSP), and a large *electromagnetic simulation* (EM) application [7].

3.1. Architectural Characteristics

3.1.1. The Shared-Memory KSR-1. The KSR-1 [1], 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 (20 MHz). 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 basic structure of the KSR-1 is the slotted ring, with a ring bandwidth divided into a number of slots circulating continuously through the ring. A standard KSR-1 ring has 34 message slots, 32 of which are designed for the 32 processors and the remaining 2 slots used by the directory cells connecting to the next ring level. Each slot can be loaded with a packet made up of a 16-byte header and a 128-byte subpage (the basic data transfer unit in the KSR-1). A processor waits for an empty slot to transmit a message. A single bit in the header of the slot identifies it as empty or full as the slots rotate through a ring interface of the processor.

3.1.2. The Connection Machine CM-5. The CM-5 [2] is the newest member of the Thinking Machines 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 or 40 MHz, 32 or 128 MB 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.

The parallel vector units on the CM-5 essentially make it a hybrid between vector and parallel architectures. Parallel variables are distributed among physical vector units. Memory layout of parallel variables onto the vector units is handled by system software, but may be overridden by the user.

All communication between physical CM-5 nodes is via packet-switched message passing on either the data or the control interconnection networks. The data network uses a fat-tree topology designed for low-latency communication of shorter messages. Global broadcasting, synchronization, and reduction operations are performed by hardware in the control network.

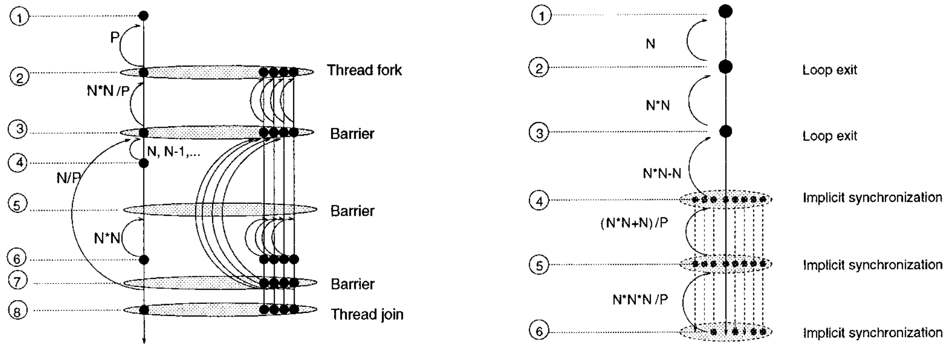

FIG. 2. The thread graph representation of the GE programs.

3.2. Application Characteristics

3.2.1. Gauss Elimination. The Gauss elimination algorithm consists mainly 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 Gauss elimination (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 synchronization point for the main thread to do the pivot. To make full 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 the left part of Fig. 2, 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 the right part of Fig. 2).

3.2.2. All Pairs Shortest Path. 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 mainly represented by a thread cluster shown in Fig. 3. In Fig. 3, P is the number of processors employed, and N is the size of the matrix.

3.2.3. The Electromagnetic Scattering Application. This application simulates electromagnetic (EM) scattering from a conducting plane body. 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 well-known moment method [7] is used for the numerical model and simulation. First, the loaded slots are imitated. Second, an equivalent admittance matrix of region "A" is calculated by using the pulse basis mode function expansion. Then intermediate results of the excitation vector and coefficient vectors are obtained based on the first two computations. At this stage the resulting EM scattering field is thus simulated by computing a large linear system called the EM strength matrix. There are four parameters representing characteristics of the equivalent magnetic current originally formed by the penetrating incident EM field. These parameters are used as mode function expansions and the number of pulse functions in the moment method. Another parameter used for visualization purposes is the number of grid points for discretizing the visualized EM scattering field. The parameters have a di-

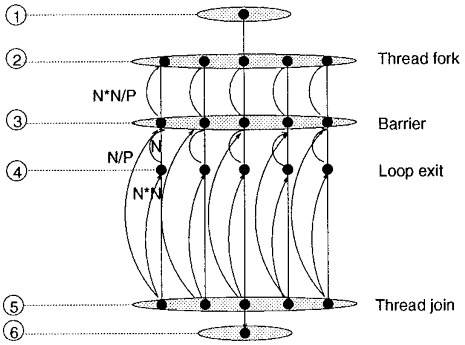

FIG. 3. A thread graph representation of the APSP program on the KSR-1.

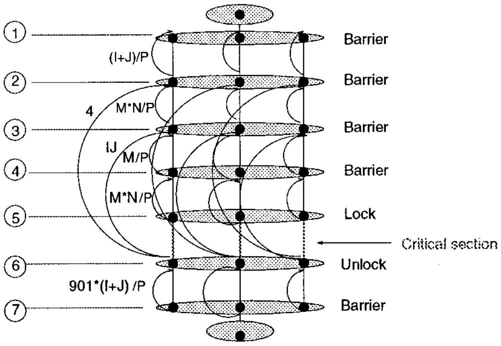

FIG. 4. A thread graph representation of the EM simulation program structure on the KSR-1.

rect impact on the computational requirements and simulation resolution of the moment method. For detailed information on the numerical method and implementations, the interested reader may refer to [7, 16].

The shared-memory implementation of the EM program is constructed mainly by a thread cluster (see Fig. 4). All edges except edge ⟨5, 6⟩ are loops. Edge ⟨5, 6⟩ corresponds to a critical section. At the end of each segment, there is a barrier. The input parameters of the EM simulation are I, J, M, and N, which determine the problem size of the simulation. The ranges of I and J we studied are from 5 to 10, and the ranges of M and N are from 20 to 256. For the data-parallel version of the EM application, instead of 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. VALIDATION AND PREDICTION RESULTS

In this section, we first give some simple metrics to evaluate the effectiveness of the performance prediction method. Then, we report the prediction and validation results using the three applications on two parallel machines.

4.1. The Effectiveness of a Performance Prediction Method

A major purpose of performance prediction of a complex system is to use as little time as possible while obtaining reasonably high accuracy.

Let $T_p^{\text{actual}}(A, \chi, N)$ denote the actual parallel execution time of application A with its parameter χ on N processors, let $T_{\text{sample}}(A)$ denote the time spent to gather the sample data, and let $T_{\text{estimate}}(A, \chi, N)$ denote the time spent to estimate the parallel execution time excluding the time to obtain the sampling data. We define the **conservative prediction ratio** as

$$R_{\text{conservative}}(A, \chi, N) = \frac{T_p^{\text{actual}}(A, \chi, N)}{T_{\text{sample}}(A) + T_{\text{estimate}}(A, \chi, N)}. \quad (4.8)$$

We call Eq. (4.8) conservative because, in practice, $T_{\text{sample}}(A)$ is used to predict the performance of an application with a wide range of application parameters. In such cases, the time spent to measure the empirical data can be neglected.

Let $T_p(A, \chi, N)$ denote the estimated parallel execution time of the application A with parameter χ on N processors. The *error* of a prediction method is defined as

$$E(A, \chi, N) = \left| \frac{T_p^{\text{actual}}(A, \chi, N) - T_p(A, \chi, N)}{T_p^{\text{actual}}(A, \chi, N)} \right|. \quad (4.9)$$

The average error, $\bar{E}$ (A, X, N), which is the average error on a set of measured points when N processors are involved, is defined as

$$\bar{E}(A, X, N) = \frac{\sum_{\chi \in X} E(A, \chi, N)}{|X|}, \quad (4.10)$$

where $|X|$ is the number of points measured. In practice, the measured points should be distributed as evenly as possible.

4.2. Performance Prediction on the KSR-1

4.2.1. The GE Program on the KSR-1. In this section we compare the predicted and measured parallel execution times of the GE program on the KSR-1. To measure the empirical data, the sizes of the sample runs used were N = 170 and N = 410 on 1 processor, N = 210 and N = 370 on 2 processors, N = 240 and N = 400 on 8 processors, N = 160 and N = 320 on 16 processors, N = 120 and N = 480 on 24 processors, and N = 192 and N = 240 on 48 processors.

Figure 5 compares the predicted and measured parallel execution times of the GE program. The application parameter N is scaled from 50 to 800 on 1 processor, from 50 to 1000 on 2 processors, from 100 to 2000 on 8 processors, from 100 to 1600 on 16 processors, and from 100 to 2400 on 24 and 48 processors. The left figure of Fig. 5 reports the validation results on 1, 8, and 24 processors while the right figure reports the results on 2, 16, and 48 processors.

Table II reports the effectiveness of the semi-empirical approach in predicting the performance of the GE program. From Table II, we find that, in most cases, the average errors are less than 10%, and the greatest average error is only 12.12%. Table II also shows good conservative prediction ratios.

4.2.2. The APSP Program on the KSR-1. Figure 6 reports the predicted and measured parallel execution times

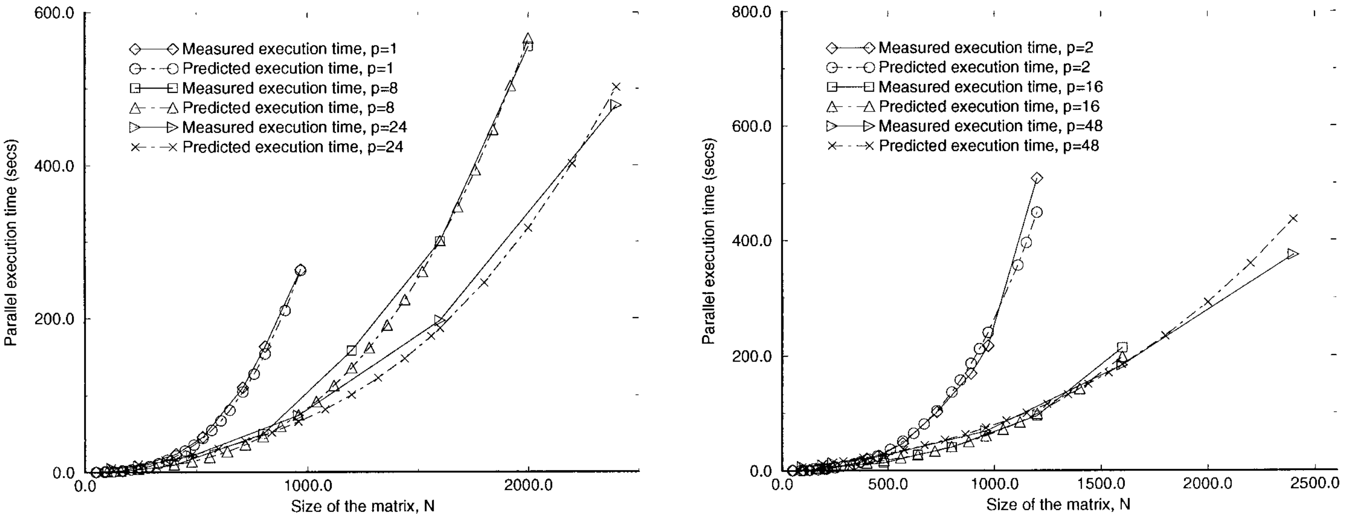

FIG. 5. Predicted and measured parallel execution times of the GE program on the KSR-1.

of the APSP program on the KSR-1. The numbers of processors are 2, 8, 16, 24, and 48, respectively, and the sample data are collected when the application parameter is set in a way similar to that for the GE program, that is, $N = 100$ and $N = 200$ on 2 processors, $N = 96$ and $N = 160$ on 8 processors, $N = 160$ and $N = 400$ on 16 processors, $N = 240$ and $N = 480$ on 24 processors, and $N = 192$ and $N = 240$ on 48 processors.

Table III reports the effectiveness of the semi-empirical approach in predicting the performance of the GE and APSP programs on the KSR-1. Table III shows good conservative prediction ratios and high precisions.

4.2.3. The EM Program. To predict parallel execution times of the EM program, we collected the sample data by setting the application parameters I and J to 5 and M and N to 20.

The predicted and measured parallel execution times of the EM program on the KSR-1 are compared in Fig. 7. In the left figure, parameters I and J are fixed at 5, and parameters M and N are scaled from 20 to 256. The numbers of processors involved are 10, 20, and 60, respectively. In the right figure, parameters I and J are further increased to 10, and all other conditions remain the same with the left figure. Like that of the GE and APSP programs, in Fig. 7 sample data on the same number of processors are used to predict the parallel execution time and to capture effects of remote memory accesses and cache coherence activities. The predicted results reasonably matched the actual execution times.

To show the effectiveness of the semi-empirical approach in this case, Table IV summarizes the conservative prediction ratios and average errors for different application parameters and machine sizes. We notice that the maximum average error of the predicted results is only about 6%. In the average case, the deviations are around 5%.

4.3. Performance Prediction on the CM-5

4.3.1. The GE Program. Figure 8 compares the predicted and measured parallel execution times of the GE program on the CM-5. The numbers of processors involved are 32, 64, and 128 processors operating in the scalar mode, and 128 processors operating in the vector mode, respectively. The application parameter N was set to 64 and 160 when sample data were collected. Because the program

TABLE II
Effectiveness of the Semi-Empirical Approach in Predicting the Performance of the GE Program on the KSR-1

No. of processors	Average error	Prediction ratio (average)	Prediction ratio (max)
1	12.12%	5	10
2	9.98%	8	28
8	5.22%	17	43
16	8.41%	5	16
24	9.32%	7	18
48	9.41%	6	17

TABLE III
Effectiveness of the Semi-Empirical Approach in Predicting the Performance of the APSP Program on the KSR-1

No. of processors	Average error	Prediction ratio (average)	Prediction ratio (max)
2	3.18%	34	106
8	5.00%	21	71
16	6.21%	21	60
24	6.00%	11	30
48	11.98%	5	14

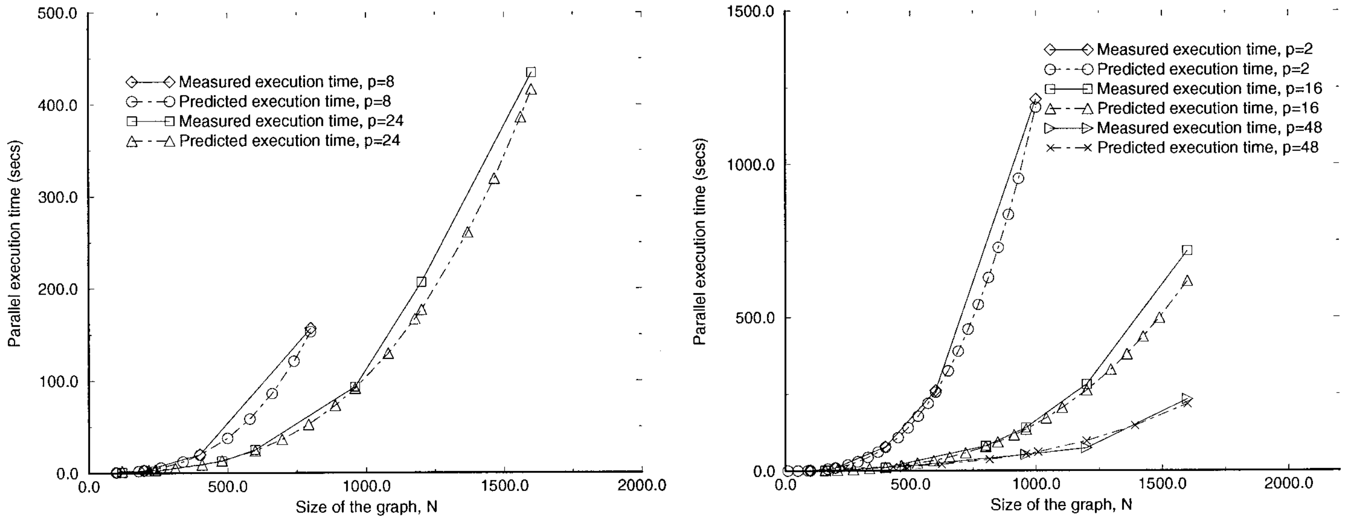

FIG. 6. Predicted and measured parallel execution times of the APSP program on the KSR-1.

structure abstraction of the GE program is fairly detailed, the predicted results are fairly precise. Table V gives the effectiveness of the semi-empirical approach in predicting the parallel execution time of the GE program on the CM-5. An interesting phenomenon is that when the processors operate in the vector mode, the predicted results are more precise when the same number of processors is involved. This is because the effects of global communication are less significant and more computation can be conducted in each node locally.

4.3.2. The EM Program. Since the structure of the data-parallel program is highly regular, in this section a higher-level abstraction of the program is used. We find that the dominant number of operations of the EM program on the CM-5 are carried out on parallel data sets with sizes of $I + J$, $M \times N$ and M, and some other data sets of constant sizes. Because the parallel fragments of the EM program are much more coarse-grained than those of the GE program, we neglected the synchronization and vector startup overheads, and so the parallel execution time of the EM program is modeled as

$$T_P = \frac{I+J}{\min(P, I+J)} c_{I+J} + \frac{MN}{\min(P, MN)} c_{MN}, + \frac{M}{\min(P, M)} c_M + \frac{c_{WAC}}{\min(P, \text{wac})}, \tag{4.11}$$

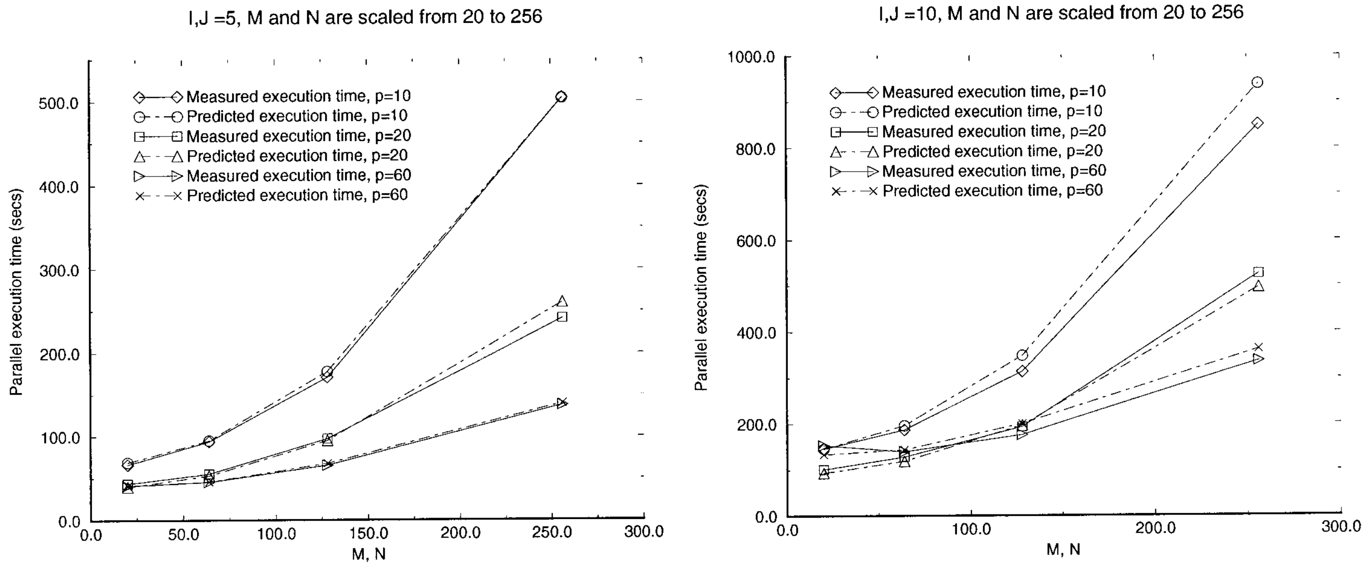

FIG. 7. Predicted and measured parallel execution times of the EM program on the KSR-1.

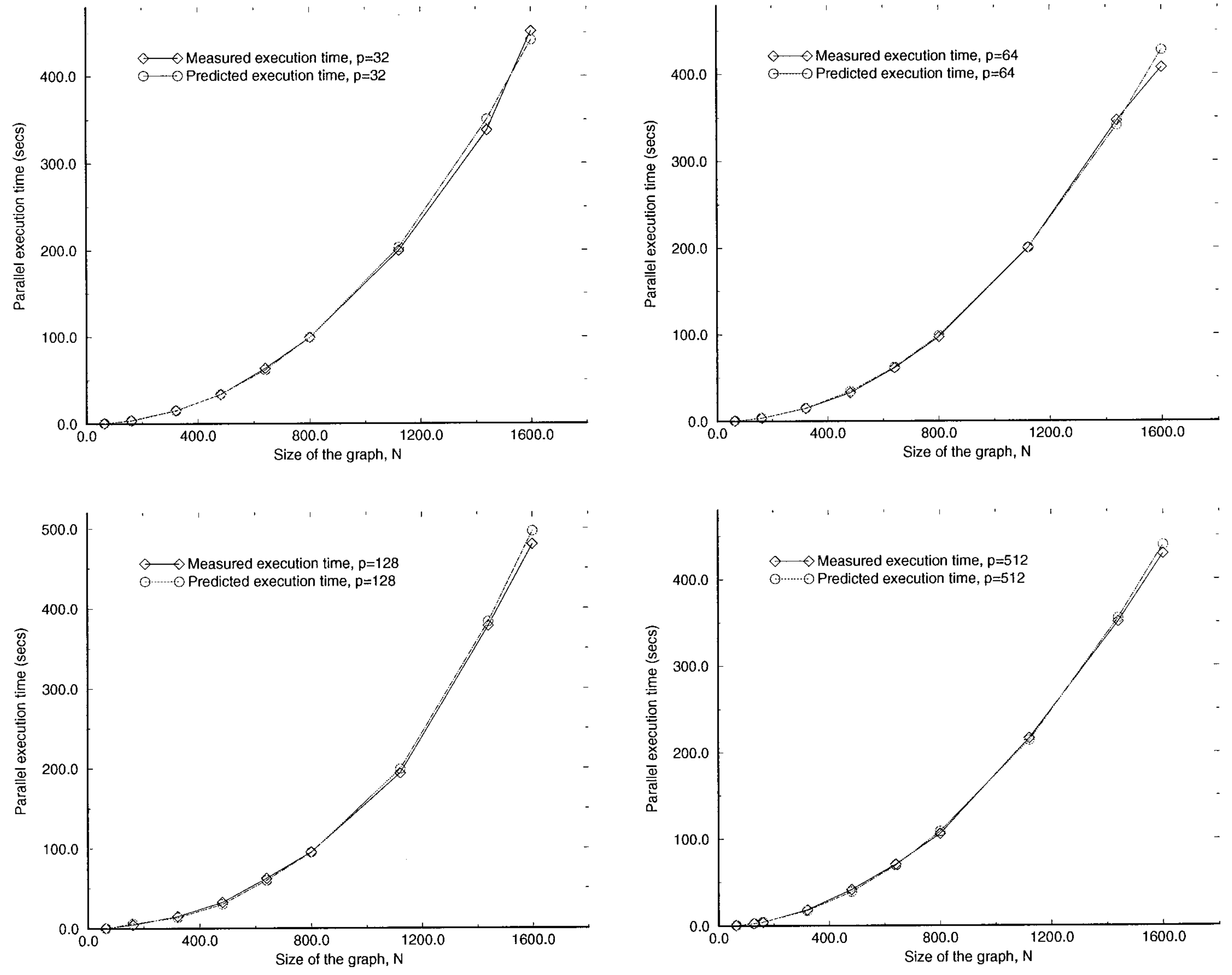

FIG. 8. Predicted and measured parallel execution times of the GE program on the CM-5.

where P is the number of processors involved, c_{I+J}, c_{MN}, and c_M are the normalized overall elapsed times of all operations on the corresponding data sets, c_{WAC} is used to approximate the data sets whose sizes are independent of application parameters, and wac stands for the weighted average constant size and is used to approximate the effects of operations on data sets of constant sizes.

Because the program abstraction is much greater than that of the GE program, the precision of prediction is not as good as with the GE program. For example, the maximum average error of the predicted results is up to 14.9%. Table VI summarizes the effectiveness of the semi-empirical approach in predicting the performance of the EM program on the CM-5, and Fig. 9 compares the predicted and measured parallel execution times.

TABLE IV
Effectiveness of the Semi-Empirical Approach in Predicting the Performance of the EM Program on the KSR-1

No. of processors	Average error	Prediction ratio (average)	Prediction ratio (max)
10	4.48%	4	13
20	5.69%	4	12
60	5.30%	3	8

TABLE V
Effectiveness of the Semi-Empirical Approach in Predicting the Performance of the GE Program on the CM-5

No. of processors	Average error	Prediction ratio (average)	Prediction ratio (max)
32	2.70%	33	101
64	2.55%	35	97
128	9.0%	31	94
128 (vector)	2.92%	32	86

TABLE VI
The Average Error of the Semi-Empirical Approach in Predicting the Performance of the EM Program on the CM-5

Application parameter	Average error
$I, J = 10; M, N = 128$	14.9%
$I, J = 10; M, N = 256$	5.94%

4.4. Summary of the Validation

- If the program abstraction is sufficiently detailed (for instance, like the thread graph representations of the shared-memory version of the EM or the shared-memory and data-parallel versions of GE) the predicted results of the semi-empirical method can be reasonably precise.
- Tables IV and V indicate that the larger the machine size, the smaller the conservative prediction ratio. There are two reasons for this. First, when the sample data are collected, the problem size must be tuned according to the machine size, so that each processing node can have enough workload to make sure that the sample data reflect the nature of the computation and communication. Second, the actual execution time will be shorter on a larger machine for the same application parameter.
- The larger the computation complexity of a program, the better prediction ratio we would expect. A better prediction ratio can also be expected for programs with smaller inherent parallelism. This is because the larger the computational complexity or smaller inherent parallelism of a program, the longer the program will execute as the application parameter scales.
- The conservative prediction ratio, or the time reduction, can be as high as tens or hundreds when the problem size is moderately large. In practice, when the application parameter is large, the conservative prediction ratio can be proportionally high. The conservative prediction ratio is a conservative measure. In reality, the same sample data can be used to predict parallel execution times of an application with the application parameter scaled in a wide range. The real prediction ratio is the summation of the total individual conservative prediction ratios when predicting the performance of the application with different application parameters.

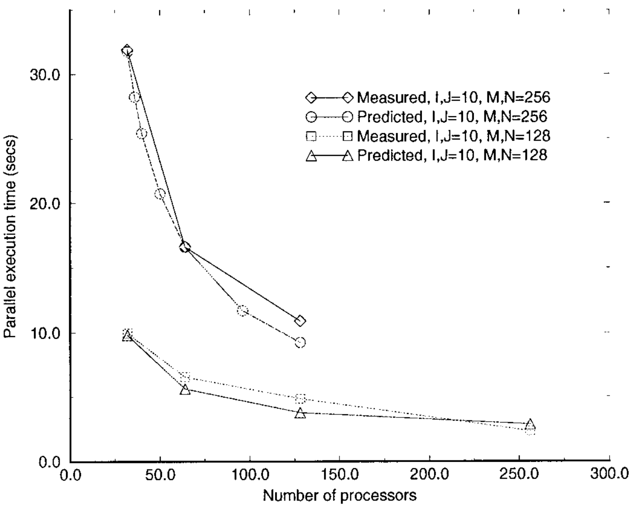

FIG. 9. Predicted and measured parallel execution times of the EM program on the CM-5.

5. DISCUSSION

We have attempted to develop a fairly complete representation of computation and communication demands of a parallel application and to achieve a good balance between using analytical techniques and using experimental measurements in predicting multiprocessor performance. In the higher-level model component, a graphical tool is used to model the communication and synchronization structure of an application. The time complexity of graphical algorithm **G** is "moderate" as a function of the number of events that must be traversed. In the lower-level model, the execution behavior of loops with a small number of iterations is used to estimate the performance of the loops with a larger number of iterations. Dynamic system and application effects are captured with measurements. For the applications and the ranges of application parameters we have studied, this methodology demonstrates reasonable prediction accuracy and good time reduction.

The following factors will affect the accuracy of our predictions in practice:

- dynamic system effects, which include network contention, remote memory accesses, and operating system intervention,
- dynamic application factors, such as the conditionals, and data-dependent computation,
- the choice of appropriate χ_0 and N_0,
- the disturbance and inaccuracy in measuring the empirical data,
- how accurately $\mathscr{F}(ss, \chi, N)$ can be estimated and the effort involved in the estimation, and
- automatic program restructuring by the compiler that causes incomplete and implicit parallelism.

To show how dynamic system effects can affect prediction accuracy, we fix the application parameters and use empirical data obtained from systems of different sizes to predict performance. Figure 10 presents the predicted performance of the EM simulation program with fixed application parameters, and using sample data collected from sample runs on different numbers of processors. From Fig. 10, we observe much larger deviations when empirical data on a different number of processors are used. However, the results also showed that the actual execution time of the EM program is slightly longer than the predicted execution times when the empirical data on a smaller number of processors are used and slightly shorter than the

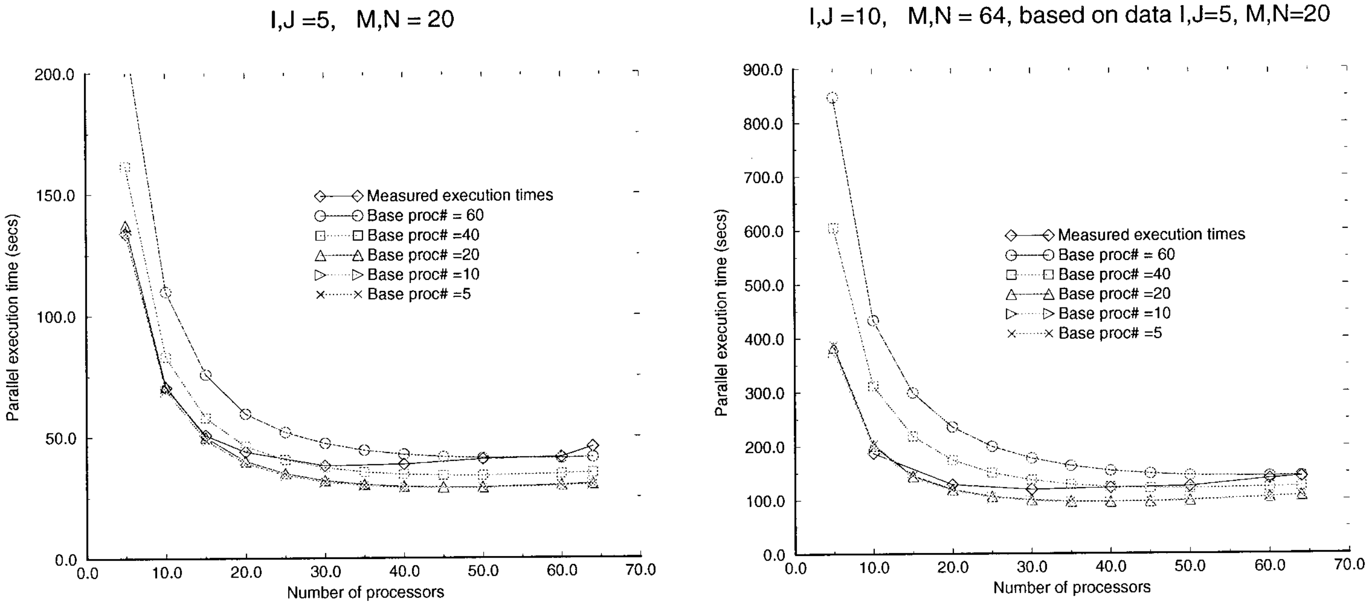

FIG. 10. Predicted and measured parallel execution times on the KSR-1.

predicted execution time when sample data on a larger number of processors are used. This indicates that it is possible to estimate the upper and/or lower bound(s) of the performance if the sample data from the same number of processors are not available.

As to the dynamic application effects, because the performances of loops with a small number of iterations are used to estimate the performances of ones with a larger number of iterations, the effects of data-dependent computations inside the loops should be partially implied in the empirical data. While it is very hard to fully validate this assumption, for the applications we have studied, we do observe quite satisfactory estimation. However, the accuracy is highly application dependent and will also depend on how we choose the sizes of the sample runs. In addition to the dynamic application effects, some important factors that are highly dependent on the choice of χ_0 include (1) the effect of the memory hierarchy, the effect of caching when χ_0 is small scale, and the effect of paging when χ is in large scale and (2) the effect of load imbalance. In the validation section, we have carefully chosen χ_0 to ensure a perfectly balanced workload to simplify our calculation. The effect of the memory hierarchy is not significant for the applications and problem sizes we studied. To capture the effects of memory hierarchies, a possible solution is to separate problem sizes into different ranges—the range that the data can be completely held in the caches, the range that the data can be held in the local memory—and the range that the data must be swapped in and out of secondary memory—and measure empirical data for each range. This may decrease the time reduction in estimating performance. An interesting research direction motivated by this is how to extract and use application- and architecture-independent empirical data.

Other factors that can cause inaccuracies in the prediction results are the instrumentation perturbation in measuring the empirical data and the compiler transformations that cannot be captured by just analyzing the source code. This includes the effect of incomplete parallelism of the compiler-generated code. So far we have only considered programs with explicit parallelism, and we use actual measurements to capture the effect of incomplete and implicit parallelism. To reduce the instrumentation perturbation, we should insert as little instrumentation code as possible. To capture the transformations conducted automatically by the compiler, three possible solutions are (1) to select important instrumentation points only (our current approach), (2) incorporate some analysis tools to capture structures of implicit and/or incomplete parallelism generated by the compiler, and (3) to analyze and to instrument the transformed code. The third solution is not economical unless it can be conducted automatically.

As to the estimation of $\mathcal{T}(ss, \chi, N)$, for large categories of applications such as numerical problems and scientific applications, $\mathcal{T}(ss, \chi, N)$ is often a simple (polynomial) function of χ and N. For applications with more dynamic execution structures, program slicing may be an effective way to estimate $\mathcal{T}(ss, \chi, N)$. However, the effectiveness of doing this is highly dependent on the overhead involved, and it is also highly dependent on the application programs.

There are several limits of our work at the current stage. First, we assume that a fairly "complete" program is available for measurements. (Although this limitation can be partially offset by implement parts of an application by demand, this will inevitably add more burden and introduce more inaccuracy.)

Second, to estimate program performance accurately with P processors requires one or more measurements of smaller inputs on a system of similar scale. In our future work, we plan to extract more information from the measurement and project changes of overheads based on more accurate and detailed system and application models.

Finally, we have only studied applications with deterministic communication structures. For applications with irregular execution structures, both prediction accuracy and time reduction can be compromised. There are three restrictions on the programs we can model. 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 parameter and system size should be easily determined by static analysis or by program slicing with small overhead. Finally, the change of application parameter should not significantly change the communication pattern of the program.

6. CONCLUSION

We present a performance prediction methodology based on both analytical models and experimental measurements. This approach allows an application user to predict the performance of a large program running on a large multiprocessor system based on a set of relatively small sample performance results. The methodology is based on a two-level hierarchical model. A proposed thread graph is used to characterize the application programs. The overhead patterns inherent in the programs in terms of communications and synchronizations are classified and treated differently according to their underlying semantics. Instead of using approximations and analytical models for the entire architecture model, some important and implicit architecture parameters are obtained from experiments on specific architectures. We believe that this execution-aided approach provides powerful and effective support for practical performance predictions, because it is impossible to capture all the important architecture effects using analytical techniques only. This is especially true for those architectures facilitated with implicit communications and dynamic scheduling schemes, such as shared-memory and data-parallel systems. Our prediction results were validated using experimental results of application programs on the KSR-1 and on the CM-5. Our validation results indicate that if the program abstraction is sufficiently detailed, the average error of the predicted results can be very small—around 7% in most cases. Even for program abstraction that is not so detailed, the maximum average error is still within 15%. In addition, the conservative prediction ratio or the time reduction can be around hundreds or even thousands for a program of moderate size.

Limitations do exist. Some augmentations might be needed for the thread graph to model application programs with a more dynamic nature. For example, for some heuristic algorithms, small changes in application parameters can significantly change the execution structure of the program. Branch statements would also cause inaccuracies in the prediction results. There are three restrictions to the programs we can model at present. 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, the change of application parameters should not significantly change the communication pattern of the program.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for providing important suggestions and comments to improve the technical quality, clarity, and readability of the paper. We appreciate our colleagues S. Dykes and N. Wagner for carefully reading the paper and contributing many helpful comments.

REFERENCES

1. Kendall Square Research. *KSR-1 Technology Background.* 1992.
2. Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary.* 1993.
3. Adve, Vikram S. Analyzing the behavior and performance of parallel programs. Ph.D. thesis, Department of Computer Sciences, University of Wisconsin–Madison, 1993.
4. Clement, M. J., and Quinn, M. J. Analytical performance prediction on multicomputers. *Supercomputing 93.* IEEE Computer Society Press, 1993, pp. 886–894.
5. Fahringer, T., and Zima, H. A static parameter based performance prediction tool for parallel programs. *Proceedings of 1993 International Conference on Supercomputing.* ACM Press, 1993, pp. 207–219.
6. Kapelnikov, A., Muntz, R. R., and Ercegovac, M. D. A modeling methodology for the analysis of concurrent systems and computations. *J. Parallel Distrib. Comput.* **6** (1989), 568–597.
7. Lu, Y., *et al.* Implementation of electromagnetic scattering from conductors containing loaded slots on the Connection Machine CM-2. *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing.* SIAM, 1993, pp. 216–220.
8. Mak, V. W., and Lundstrom, S. F. Predicting performance of parallel computations. *IEEE Trans. Parallel Distrib. Systems* **1,** 3 (July 1990), 257–269.
9. Menasc̃e, D. A., and Barroso, L. A. A methodology for performance evaluation of parallel applications on multiprocessors. *J. Parallel Distrib. Comput.* **14,** 1 (1992), 1–14.
10. Menasc̃e, D., Noh, S. H., and Tripathi, S. K. A methodology for performance prediction of massively parallel applications. *Proc. of the Fifth IEEE Symposium on Parallel and Distributed Processing.* IEEE Comput. Soc. Press, 1993, pp. 250–257.
11. Mohan, J. Performance of parallel programs: Model and analyses. Ph.D. dissertation, Carnegie–Mellon University, 1984.
12. Sun, X.-H., and Zhu, J. Performance prediction of scalable computing: A case study. *Proceedings of the 28th Hawaii International Conference on System Sciences,* Vol. II. IEEE Comput. Soc. Press, 1995, pp. 456–465.
13. Thomasian, A., and Bay, P. F. Analytic queueing network models for parallel processing of task systems. *IEEE Trans. Comput.* **C-35,** 12 (1986), 1045–1054.
14. Tsuei, Thin-Fong, and Vernon, Mary K. Diagnosing parallel program speedup limitations using resource contention models. *Proceedings of the 1990 International Conference on Parallel Processing, August 13–17, 1990.* Pp. I-185–I-189.

15. Vrsalovic, D. F., Siewiorek, D. P., Segall, Z. Z., and Gehringer E. F. Performance prediction and calibration for a class of multiprocessors. *IEEE Trans. Comput.* **37,** 11 (Nov. 1988), 1353–1365.

16. Zhang, X., and Sun, L. Comparative evaluation and case studies of shared-memory and data-parallel execution patterns on KSR-1 and CM-5. *Sci. Programming,* to appear.

17. Zhang, X., Yan, Y., and He, K. Evaluation and measurement of multiprocessor latency patterns. *Proceedings of the 8th International Parallel Processing Symposium.* IEEE Comput. Soc. Press, 1994, pp. 845–852.

18. Zhang, X., Xu, Z., and Sun, L. *Performance Modeling and Implications of the Fat-Tree Networks and the CM-5 Architecture.* Technical Report, High Performance Computing and Software Laboratory, The University of Texas at San Antonio, 1994.

ZHICHEN XU received his B.S. in computer science from Fudan University, China, in 1987 and his M.S. in computer science from the University of Texas at San Antonio in 1995. He is currently working toward his Ph.D in computer science at the University of Wisconsin at Madison. His research interests are parallel programming languages, parallel and distributed systems, and parallel performance measurement tools. Xu is a member of the ACM.

XIAODONG ZHANG is an associate professor of computer science at the University of Texas at San Antonio, where he directs the High Performance and Computing and Software Laboratory. His research interests are parallel and distributed computation, computer system performance evaluation, and scientific computing. Zhang received his Ph.D. in computer science from the University of Colorado at Boulder in 1989. He is a senior member of the IEEE and the current Chair of the IEEE Technical Committee on Supercomputing Applications.

LIN SUN received his B.S. in computer science from Zhejiang University, China, in 1986 and his M.S. in computer science from the University of Texas at San Antonio in 1994. He has been a system analyst working for US West Advanced Technology since 1994. His research interests are software development and computer system performance evaluation.

Received July 15, 1994; revised January 10, 1996; accepted August 13, 1996