Parallel Program Performance Prediction Using Deterministic Task Graph Analysis

Vikram S. Adve
University of Illinois at Urbana-Champaign
and
Mary K. Vernon
University of Wisconsin-Madison

In this paper, we consider analytical techniques for predicting detailed performance characteristics of a single shared memory parallel program for a particular input. Analytical models for parallel programs have been successful at providing simple qualitative insights and bounds on program scalability, but have been less successful in practice for providing detailed insights and metrics for program performance (leaving these to measurement or simulation). We develop a conceptually simple modeling technique called deterministic task graph analysis that provides detailed performance prediction for shared-memory programs with arbitrary task graphs, a wide variety of task scheduling policies, and significant communication and resource contention. Unlike many previous models that are stochastic models, our model assumes deterministic task execution times (while retaining the use of stochastic models for communication and resource contention). This assumption is supported by a previous study of the influence of non-deterministic delays in parallel programs.

We evaluate our model in three ways. First, an experimental evaluation shows that our analysis technique is accurate and efficient for a variety of shared-memory programs, including programs with large and/or complex task graphs, sophisticated task scheduling, highly non-uniform task times, and significant communication and resource contention. The results also show that the deterministic assumption is crucial to permit accurate and yet efficient analysis of these programs. Second, we use three example programs to illustrate the predictive capabilities of the model. In two cases, broad insights and detailed metrics from the model are used to suggest improvements in load-balancing and the model quickly and accurately *predicts* the impact of these changes. In the third case, the model provides novel insights into the impact of program design changes that improve communication locality as well as load-balancing, via new (but general-purpose) metrics. Finally, we present results from a comparison of our model and representative stochastic models, and use these to characterize the conditions under which a deterministic model or stochastic models would be appropriate.

Categories and Subject Descriptors: I.6.5 [**Model Development**]: Modeling methodologies; C.4 [**Performance of Systems**]: Modeling techniques; G.3 [**Probability and Statistics**]: Stochastic Processes

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: Analytical model, deterministic model, parallel program performance prediction, queueing network, shared memory, task graph, task scheduling.

This work was supported in part by the National Science Foundation (CDA-9024618, CDA-8920777, CCR-9024144, and EIA-00-93426), by DARPA Contract No. N66001-97-C-8533, and by an IBM Graduate Fellowship.

Author's addresses: Vikram S. Adve, Computer Science Department, 1304 W. Springfield Ave., Urbana, IL 61801 (`vadve@cs.uiuc.edu`). Mary K. Vernon, Computer Sciences Department, 1210 West Dayton St., Madison WI 53706 (`vernon@cs.wisc.edu`).

1. INTRODUCTION

This paper considers analytical techniques for evaluating the performance and design alternatives of a shared memory parallel program executing on a shared memory system either stand-alone or with a scheduler that minimizes interference from other programs. The focus of this work is on techniques that can be used for a *detailed* quantitative understanding of parallel performance issues, and for predicting the impact of program design alternatives that could improve parallel performance.

An appropriate analytical model[1] for detailed program performance prediction can complement measurement and simulation techniques, and can be of significant practical value for three reasons. First, it can provide an abstraction description of program behavior that is useful for gaining insight into key program performance issues. Second, it can provide a more efficient approach for exploring details of program performance. Third, it can be used to predict the performance of a given program on future systems or system configurations, and to predict the performance impact of program design changes *before* implementing the changes in the code.

Analytical performance models for parallel programs have been widely used for obtaining qualitative insights and bounds on scalability of various parallel algorithms based on key parameters such as input and system size [Amdahl 1967; Gustafson 1988; Culler et al. 1993; Alexandrov et al. 1995; Frank et al. 1997]. In contrast, previous analytic models for more detailed analysis of parallel program execution time [Dubois and Briggs 1982; Mohan 1984; Kruskal and Weiss 1985; Thomasian and Bay 1986; Kapelnikov et al. 1989; Ammar et al. 1990; Mak and Lundstrom 1990; Vrsalovic et al. 1988; Cvetanovic 1987; Tsuei and Vernon 1990; Harzallah and Sevcik 1995; Xu et al. 1996; Jonkers et al. 1995] have been less successful, and are also less well understood. In practice, these models either require a specialized model derivation for each new parallel program, or, as explained in section 6, they are restricted to particular program synchronization structures, task scheduling algorithms, or task execution time distributions that can be analyzed. The restricted models either don't apply or have unknown accuracy for many parallel applications that one would like to be able to analyze. Thus, practical tools for detailed performance analysis of parallel programs are currently based on measurement or simulation rather than on analytical modeling.

One of the key challenges in modeling a parallel program is how to represent variability in task execution times so as to permit tractable analysis of program performance. Many previous models for detailed performance prediction are stochastic models, i.e., using a stochastic representation of task execution times. This assumption can make it quite challenging to estimate average synchronization costs, except for programs with simple fork-join synchronization (i.e., alternating serial and parallel phases) and relatively simple task scheduling. In Section 2.2, we use results from a previous study of the impact of variability in shared memory programs to motivate a simpler approach based on using deterministic task execution times in a detailed performance prediction model. (This approach has also been used in a few previous and fairly accurate, but restricted, models [Vrsalovic et al. 1988; Cvetanovic 1987; Tsuei and Vernon 1990; Harzallah and Sevcik 1995; Xu

[1]Unless stated otherwise, we use the term "model" both for an abstract description of program behavior and for the solution technique used to compute performance metrics from that description.

et al. 1996; Jonkers et al. 1995; van Gemund 2003]).

This paper develops and evaluates a conceptually simple and efficient model for performance prediction of shared memory parallel programs, which is applicable to a wider range of programs than previous detailed models in the literature. Our model assumes deterministic task execution times (but permits stochastic techniques for estimating average communication costs and resource contention). The inputs to the model are: 1) a task graph that describes the sequential tasks and synchronization behavior of an application (similar to that used by many previous models); 2) a description of the task scheduling algorithm used within the program to allocate tasks to threads, and 3) parameters describing the computation time, communication rates, and shared resource usage behavior of each task. In contrast to previous models, we use a solution technique based on a modified critical path analysis of program execution time that incorporates both a precise analysis of task scheduling and a mean value analysis of communication costs using a separate (usually stochastic) system model. This solution technique, which we call *deterministic task graph analysis*, applies to parallel programs with arbitrary task graphs and a wide range of static and dynamic task scheduling methods. As shown in section 5, the proposed model provides detailed insights into the impact of process synchronization and task scheduling on program performance. The model also predicts the average communication costs (including contention) incurred by individual tasks and their impact on overall execution time.

We experimentally evaluate the deterministic task graph model using realistic parallel applications executing on realistic inputs. For the applications we have examined, we find that the deterministic task graph model is very efficient to evaluate even for programs with large and complex task graphs. More importantly, it is *consistently accurate* (with typical errors of less than 10% in predicting measured execution time) because it accurately represents key details of task scheduling, the order of task execution, non-uniform task execution times, and average communication costs including contention.

We use three of the programs to demonstrate the use of deterministic task graph analysis for understanding performance bottlenecks and for predicting the impact of hypothetical modifications to existing programs. In two cases, insights and specific metrics from the model suggested improvements in load-balancing and the model quickly and accurately *predicted* the improved performance for these changes, as shown by subsequently modifying the code. In the third case, we developed further general-purpose metrics to obtain insight into the impact of sophisticated program design changes that improve communication locality as well as load-balancing. Because the key abstractions in the model are the task graph, scheduling algorithm, and system architecture, the model is best suited for predicting the impact of changes in these components, but not of changes that affect the numerical task parameters. This limitation is discussed further in the Conclusions.

Finally, one of the contributions of this research has been to evaluate representative stochastic models from the literature on a set of realistic applications, to understand the strengths and weaknesses of the stochastic models, and compare them with the deterministic task graph model. To our knowledge, none of these models have previously been evaluated using realistic applications. Section 6.3 uses these experimental results to classify programs into three categories, based on

key characteristics that determine when existing stochastic or deterministic models might be preferable.

Briefly, we find that stochastic models for programs with simple fork-join synchronization (i.e., alternating serial and parallel phases), are simple, accurate, and highly efficient, but are only accurate for restricted task scheduling or task time distributions. In contrast, we find that for the other two classes of programs, which include four of the five pograms studied in our work, existing stochastic models appear inaccurate or impractical. In particular, stochastic models that allow complex synchronization and/or task scheduling algorithms proved expensive or impractical to solve for realistic programs. These models also showed poor or inconsistent accuracy, primarily because they all assume exponential task times for enabling tractable analytic solutions.

The rest of this paper is organized as follows. In Section 2, we discuss key terms and concepts used in the paper and then motivate our modeling approach. In Section 3, we describe the deterministic task graph model. In Section 4, we evaluate the accuracy, efficiency and applicability of our model. In Section 5, we illustrate how our model can be used for evaluating program design tradeoffs. In Section 6, we first review previous analytical models, and then present an experimental comparison of representative previous models with our model. Section 7 presents a summary of the strengths and limitations of the deterministic model and suggests directions for future research.

2. PRELIMINARIES AND MOTIVATION FOR A DETERMINISTIC MODEL

To provide a framework for discussing parallel program performance models, Table I defines our usage of a few key terms and concepts. Figure 1 illustrates some of these terms using the task graphs for five shared-memory programs. These programs are described in more detail in Section 4.

The *task graph* provides an abstract but precise representation of the parallelism and synchronization in a program, for a particular input. Perhaps our most important goal in choosing these definitions is that, ideally and whenever possible in practice, the task graph should be a representation of the inherent parallelism in the program for a particular input, *independent of the number of processes or processors that execute the program*. Therefore, we have defined the task graph to be separate from the *task scheduling function*, and we have defined a *task* to be a unit of work that is executed by a single process *in any execution* of the program for a fixed input. For example, the tasks may be the iterations of a loop, and the scheduling function would specify how the iterations are scheduled onto the processes that execute the program. In some shared memory programs, per-process initialization code would have to be represented as one task per process. These tasks are often small enough to be ignored without significant loss of accuracy. Using this simplification, the above property holds for all the task graphs in Figure 1.

The task scheduling function is important because shared-memory programs may used sophisticated static or dynamic task scheduling algorithms to achieve good load balance and locality. Common task scheduling algorithms include static allocation of loop iterations in consecutive or round-robin order, dynamic allocation from a single task queue, or more complex semi-static policies that balance dynamic load balancing and locality, such as those described for the application `LocusRoute` in

Table I. Definitions of Key Terms

Task A unit of work in a parallel program that is always executed by a single process in any execution of the program, and such that any precedence relationship between a pair of tasks only arises at task boundaries.

Task Graph A directed acyclic graph in which each vertex represents a task and each edge represents a precedence between a pair of tasks. A task can begin execution only after all its predecessor tasks, if any, complete execution.

Process A logical entity that executes tasks of a program. Also the entity that is scheduled onto processors. In a multithreaded program, this corresponds to a single thread.

Task Scheduling Function For a given set of ready tasks and a given idle process, a function that specifies which of the tasks will be executed next by that process.

Condensed Task Graph (For a program and a particular allocation of tasks to processes) A directed acyclic graph in which each vertex denotes a collection of tasks executed by a single process, and each edge denotes a precedence between a pair of vertices (i.e. all the tasks in the vertex at the head of the edge must complete before any task in the vertex at the tail can begin execution).

Fork-join Task Graph A task graph consisting of alternating sequential and parallel phases, where each parallel phase consists of a set of independent tasks and ends in a full barrier synchronization [Towsley et al. 1990].

Series-Parallel Task Graph A task graph that can be reduced to a single vertex by repeated applications of *series reduction* or *parallel reduction*: *Series reduction* combines two vertices V_1 and V_2 into a single vertex if V_1 is the only parent of V_2 and V_2 is the only child of V_1. *Parallel reduction* combines 2 vertices V_1 and V_2 into a single vertex if V_1 and V_2 have exactly the same parents, as well as exactly the same children [Hartleb and Mertsiotakis 1992].

Section 5.3.

The condensed task graph is a compact version of the task graph in which consecutive tasks executed by a single process between synchronization points are aggregated into a single vertex. This can reduce task graph size greatly for some programs, but it can only be used when the allocation of tasks to processes can be precomputed (such as for static task scheduling algorithms). It can also exacerbate the errors due to the exponential task assumption [Adve 1993], as seen in Section 6.2. Finally, fork-join and series-parallel task graphs are restricted classes of task graphs with simplified synchronization structures (the fork-join class is a subset of the series-parallel class). Many previous analytical models have been restricted to one of these classes of graphs. Figures 1 (a)–(c) are fork-join graphs, while (d)–(e) are general non-series-parallel graphs.

We believe the task graph and scheduling function together provide an appropriate level of abstraction for detailed quantitative analysis of parallel program performance. It is a less detailed representation than the actual program, yet provides sufficient information for evaluating many important program performance issues. Furthermore, most previous analytic models for detailed performance prediction are based on graph models that can be viewed as equivalent to either the task graph or the condensed task graph.

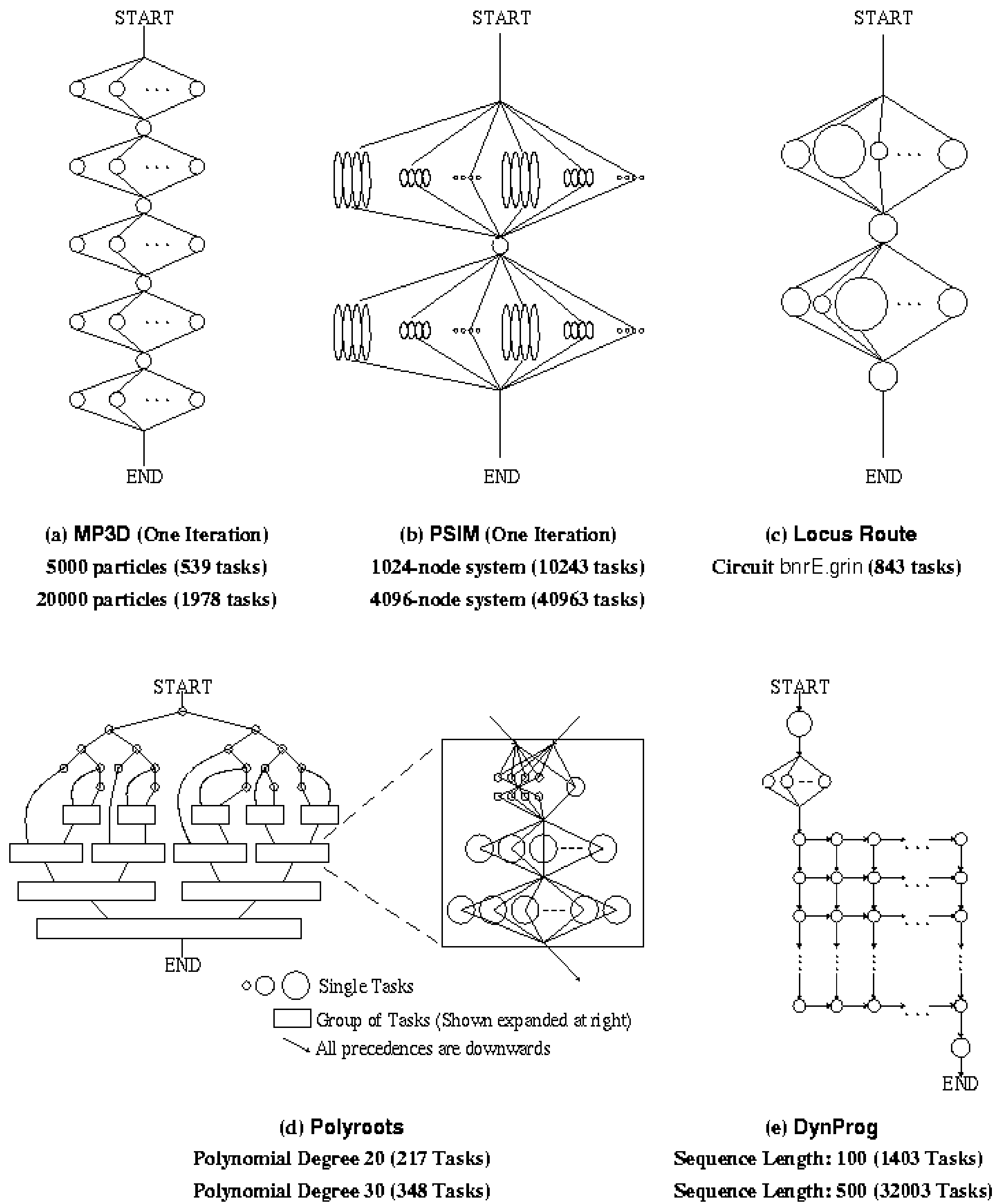


Fig. 1. Task Graphs for Five Shared Memory Applications.
`START` and `END` denote dummy tasks marking the entry and exit for each graph.

2.1 A Framework for Parallel Program Performance Prediction Models

Throughout this paper, unless stated otherwise, our discussions of previous models focuses on models for detailed performance prediction and not on simpler parametric models, as motivated in the Introduction. Section 6 and [Adve 1993] discuss in more detail how the goals of simpler models complement our work.

A number of models for parallel program performance prediction (including our own) have been constructed as two-level hierarchical models and, in fact, all the models we discuss can be cast into the same hierarchical framework. The higher-level component in this hierarchy represents the task-level behavior of the program, namely task scheduling, execution and termination, and process synchronization. Assuming individual task execution times are known, this model component computes the overall execution time of the program and perhaps other metrics as well. The individual task times may be represented either as deterministic or stochastic quantities. The key challenge in developing a good overall model usually lies in predicting synchronization costs and task scheduling behavior, particularly for programs with widely varying task times or non-fork-join task graphs.

Individual task execution times (or their statistics) are computed from the lower-level model component. This component represents system-level effects and all shared resources, e.g., communication costs, network contention, lock contention, etc. This component can be an analytical model, typically a queueing network model of the system, or can use direct measurement or simulation. The solution of this model component must account for the effect of task precedences and scheduling. For example, the most general previous stochastic models solve the queueing network for each distinct combination of tasks in execution [Thomasian and Bay 1986; Kapelnikov et al. 1989], while some models uses additional parameters computed from the higher-level model to account for this effect approximately [Mak and Lundstrom 1990; Liang and Tripathi 2000].

2.2 Motivation for a Deterministic Model

We use the terms "deterministic model" and "stochastic model" respectively for models that represent the total execution time for each task as a deterministic or a stochastic quantity. The possible reasons for choosing one or the other are discussed below. Note that stochastic task times should *not* be used to capture the behavior of a program *for different inputs*, since the statistics of these task times affect synchronization costs within a single execution. In this work, we focus on models for predicting program performance for a particular input (as represented by a single task graph). Scalability models such as Vrsalovic et al. [Vrsalovic et al. 1988] can be used to study behavior across different inputs directly. Alternatively, behavior across different inputs can be captured by constructing separate task graphs and then solving the model separately for different inputs (which is not difficult for a model that can be solved in tens of seconds or less).

A stochastic model represents a program (explicitly or implicitly) as a stochastic process in which each state is some combination of tasks in execution. Average synchronization costs in such a model represent the average across all possible sequences of task execution, and the number of possible sequences is extremely large. Therefore, except for simple task graph structures where closed-form estimates of

synchronization cost are possible, computing average synchronization costs with a stochastic model that assumes arbitrary distributions of task times can be extremely complex. Thus, all previous stochastic models we are aware of are either restricted to simple task graph structures, or assume exponentially distributed task times for analytical tractability. A quantitative evaluation of the accuracy and efficiency of some representative examples of such models showed that models in the latter class are expensive to solve even for relatively small programs, and also have poor or inconsistent accuracy due to the assumption of exponential task execution times [Adve 1993]. These results and a comparison with our model are presented in Section 6.

The potential complexity of stochastic models leads us to consider a *deterministic* model, in which the higher level model component assumes deterministic task execution times. The key advantage of the deterministic assumption is that it implies a unique execution sequence for the program, and the delay at each synchronization point in the sequence can be calculated as simply the numerical maximum of the execution times of the synchronizing processes.

A few previous analytical models assume deterministic task execution times [Vrsalovic et al. 1988; Cvetanovic 1987; Tsuei and Vernon 1990; Harzallah and Sevcik 1995]. These models have each been shown to be both accurate and efficient for several parallel programs to which they apply, and the deterministic assumption has significantly simplified the analysis in the models. However, these previous models too are only useful for programs with simple fork-join synchronization, as explained in Section 6. The previous authors also did not provide any direct justification for the deterministic assumption, which has some potential limitations.

There are two fundamental limitations of deterministic models. One limitation is that to model a parallel program phase with unequal task execution times, a deterministic model would require a detailed specification of *individual* task times, whereas a stochastic model could use a simple set of statistical parameters such as the mean and variance. Therefore, stochastic models would be preferable when only simple statistics about task times are known. Estimating such statistics can be difficult, however, for unstructured task graphs without clear parallel phases, e.g., `Polyroots` in Figure 1(d)). Furthermore, current stochastic models that permit such statistics (a mean and variance) are restricted to fork-join programs with limited types of task scheduling, as described in Section 6.

A second fundamental limitation of a deterministic model is that it cannot account for any possible variability in the execution times of *individual* tasks, which could cause the model to underestimate synchronization costs. There are three sources of such variability in individual task execution times. First, variability can arise due to multiprogramming of the processors. The impact of this variability must be estimated during the analysis (e.g., as resource contention within the system-level model), and incorporated in the higher-level model. Such analysis is outside the scope of our work, i.e., we focus on a program executing stand-alone or with a scheduler that minimizes interference from other programs (e.g., one that gives the program a dedicated partition of a multiprocessor system).

The two other sources of variability are the variability in the communication costs of the tasks due to resource contention, and inherent variability in the computation times of the tasks. Both of these have been cited as arguments for assuming non-

deterministic task times [Kruskal and Weiss 1985; Dubois and Briggs 1982].

In a recent study, we used an analytical model combined with detailed program measurements to study the impact of variable communication delays on task execution times in shared-memory parallel programs [Adve and Vernon 1993]. That study showed that in shared-memory programs with coarse-grain or fine-grain tasks, random delays due to communication and contention introduce negligible or very small variance into the execution time of a process between synchronization points. In other words, the principal effect of such random delays is to increase the mean execution time of the process, while the variance in execution time remains essentially unaffected. This holds for large-scale shared-memory systems as well as smaller single-bus systems. Intuitively, this result holds because a large number of communication delays (e.g., cache misses) occur in a typical interval between synchronization points of a shared-memory parallel program. While the individual delays may have significant variance and therefore fluctuate considerably around their mean values, the fluctuations of a large number of such delays tend to cancel each other out.

Second, some programs exhibit variability in the task computation times themselves, even for a fixed input. For example, programs with data races that significantly affect the computation may exhibit such characteristics. A stochastic model could represent variability in task computation times (as well as variability due to communication delays) but computing model input parameters that capture these sources of variability would be extremely hard. Even in such programs, a deterministic model can provide results about one particular execution and thus can be useful for program performance studies. In fact, the program `LocusRoute` exhibits this behavior, and we study the accuracy and usefulness of our model for this program is studied in Sections 4 and 5. In many such cases, however, we expect the overall performance impact to be relatively small (as with `LocusRoute`).

Based on these arguments and experiments, we hypothesize that variability in individual task execution times will produce relatively small errors in modeling accuracy, and the potential simplicity of a deterministic model would make such an assumption worthwhile. We represent each task execution time as a deterministic quantity equal to the sum of the CPU requirement of the task and the mean total overhead experienced by the task.

3. A DETERMINISTIC PERFORMANCE MODEL FOR PARALLEL PROGRAMS

In this section we propose a two-level hierarchical model in which the lower level model component is a standard, possibly stochastic, system resource usage model, but the higher level model component assumes deterministic task execution times. Section 3.2 describes the higher-level (or task-level) component of our model. Section 3.3 describes an example lower-level (or system-level) component for the Sequent Symmetry multiprocessor.

3.1 Inputs to the Model

The model inputs are defined in Table II. N and $Parents(i)$ together define the task graph. We assume without loss of generality that task 1 is the only task with no predecessors. If more than one such task exists in the program, we can add a dummy task with zero processing time as a predecessor to all such tasks.

Table II. Inputs to the Deterministic Model

Parameter	Explanation
N	Number of tasks
Parents(i), $1 \leq i \leq N$	List of direct predecessors for each task i (assume task 1 is the only task with no predecessors)
T_i, $1 \leq i \leq N$	Fixed mean CPU demand for each task i
$M_{i,j}$, $1 \leq i \leq N$, $1 \leq j \leq N_{res}$	N_{res} resource usage parameters for each task i
$Sched(L, p)$	Scheduling function: specifies which task from ready task list L (if any) is executed next by idle process p
P	Number of processors

The parameters T_i and $\{M_{i,j} : 1 \leq j \leq N_{res}\}$ together characterize the resource demands of each task, i. The parameter T_i represents the CPU demand of task i. In practice, it can also include other components of the task execution time that are fixed independent of the state of the program, if those components can be estimated separately without using the system-level model, and if the CPU is not relinquished by the task (e.g., I/O operations that do not relinquish the CPU). The set of parameters $\{M_{i,j} : 1 \leq j \leq N_{res}\}$ is the set of resource usage parameters that are used by the lower-level model. The number and type of resource usage parameters depend on the choice of system-level model used, and on the shared resources to be modeled. Some possible examples of resource usage parameters for a cache coherent shared memory system are a task's cache miss rate and the fraction of cache misses that cause write-backs for dirty cache lines.

In the table, we have represented the task scheduling algorithm in the form of a scheduling function $Sched(L, p)$ which, given a list of ready tasks, L, and an idle process, p, specifies which task, if any, will be executed next by process p. For example, for dynamic allocation from a single task queue, process p simply gets the first task in L. This definition of a scheduling function is fairly general but is not practical to use as an input representation in practice. Instead, our implementation of the model defines a common scheduling framework that, although not universal, is sufficient to describe a large class of static, semi-static and dynamic scheduling policies used in shared-memory codes today [Adve 1993]. This is simply a user convenience; the model and solution algorithm are not limited to these policies.

Henceforth, we assume that only one process per processor is used during the execution of the program, as in many parallel scientific and engineering applications today (using our definition of the term "process" in Section 2). In [Adve 1993], we discuss simple extensions to model multiple processes per processor.

3.2 The High-Level Model of Task Execution

The high-level model component in deterministic task graph analysis essentially performs three key functions:

(1) a simple graph traversal algorithm to enforce task precedences and to track the execution state of individual tasks and processes;

(2) evaluating the scheduling function to model the impact of task allocation (including task execution order) when there are fewer processors than the maxi-

mum parallelism available in the task graph; and

(3) invoking a separate lower-level (i.e., system-level) model to compute the impact of communication costs and resource contention.

Figure 2 presents the complete model solution algorithm, including the invocations of the lower level system model. Before describing the complete model, we first describe a basic but useful algorithm with no lower-level model, i.e., considering only the task graph and scheduling, and ignoring other costs such as communication and resource contention. The basic algorithm can be determined from the figure by ignoring the first highlighted step which computes the values D_i (defined below), and by assuming $D_i \equiv 0, 1 \leq i \leq N$ in the other three highlighted steps.

The Basic Model Ignoring Resource Contention

Assume that the fixed CPU demand T_i completely captures the total execution time of task i. Then, the lower-level system model is not needed. In section 4 we give two examples of programs for which this basic model is sufficient to provide accurate and detailed performance prediction on the Sequent Symmetry. In this case, for a particular number of processors and a deterministic scheduling algorithm, the program has a *unique execution sequence*, i.e. a unique sequence of times at which particular tasks begin and complete execution. Furthermore, a simple algorithm can be used to compute the sequence of task initiations and terminations as well as the *exact* execution time of the program.

Let $\mathcal{E}$ denote the set of executing tasks, L the list of ready tasks, and r_i the remaining CPU requirement of task i. We initialize r_i to T_i, $\mathcal{E}$ to contain task 1 and L to be empty. Essentially, the basic algorithm consists of repeating steps 2, 3, 4 and 5 at most N times (with $D_i \equiv 0$):

2) Delete one or more tasks with the minimum remaining CPU demand from $\mathcal{E}$.
3) Update remaining CPU demand r_i of other tasks $i \in \mathcal{E}$.
4) Find any newly ready tasks (viz., unfinished tasks whose predecessors have all completed) and add them to L.
5) For each idle process p, apply the scheduling function to determine which ready task, if any, should be scheduled on the process.

This basic algorithm, namely computing the unique execution sequence for a program, provides the basis for deterministic task graph analysis. In the extreme case of an unlimited number of processors, the algorithm simply computes the critical path in the graph, which corresponds to how programmers reason about program execution time. The existence of such a simple underlying model which is exact under the stated assumptions is a key benefit of the deterministic approach.

Even though the basic model ignores important sources of (variable) overheads such as resource contention, it can still be of practical use. Two example scenarios where it can be useful are performance tuning of programs with negligible or small communication and mutual exclusion overheads (e.g., `Polyroots` and `DynProg` in Section 5.1), and approximate prediction of the impact of changes in parallel structure and scheduling for a program, ignoring such overheads.

Key variables used in the algorithm

$\mathcal{E}$	Current set of executing tasks
L	Set of ready tasks waiting to be scheduled
r_i	Remaining CPU requirement of task i
D_i	Total overhead incurred by task i, if task i completes before other tasks in $\mathcal{E}$
ctr_i	Number of parents of task i that have not completed yet
T_{elapse}	Time until next task completion

Inputs All inputs listed in Table II

Algorithm

$r_i \leftarrow T_i$, $1 \le i \le N$ /* Remaining cpu requirement for task i */
$ctr_i \leftarrow$ #Parents of i, $1 \le i \le N$
$\mathcal{E} \leftarrow \{ 1 \}$ /* Initial set of executing tasks */
$L \leftarrow \{ \}$ /* Initial set of ready tasks waiting to be scheduled */
$T_{total} \leftarrow 0$ /* Elapsed time since start of program */

Do until $\mathcal{E}$ empty { /* At most N times; exactly N if no two tasks complete simultaneously */

1) Compute $D_i(\mathcal{E}, \{M_{i,j}\}, \{r_j : j \in \mathcal{E}\})\ \forall\, i \in \mathcal{E}$ using the lower-level system model

2) $T_{elapse} \leftarrow \min\{r_i + D_i : i \in \mathcal{E}\}$ /* Time till next task completion */
$\mathcal{C} \leftarrow \{j \in \mathcal{E} : r_j + D_j = T_{elapse}\}$ /* Set of tasks that complete next */
$\mathcal{E} \leftarrow \mathcal{E} - \mathcal{C}$

3) $T_{total} \leftarrow T_{total} + T_{elapse}$
$r_i \leftarrow r_i - T_{elapse} \times \frac{r_i}{r_i + D_i}$, $\forall i \in \mathcal{E}$

4) For each task $x \in \mathcal{C}$
  For each immediate successor c of task x
    $ctr_c \leftarrow ctr_c - 1$
    If $ctr_c = 0$ /* I.e., if all parents of task c have completed */
      $L \leftarrow L \bigcup \{c\}$ /* Add task c to list of ready tasks */

5) For each idle process p
  $j \leftarrow Sched(L, p)$
  if $j > 0$
    $L \leftarrow L - \{j\}$
    $\mathcal{E} \leftarrow \mathcal{E} \bigcup \{j\}$
}

Total Program Execution Time $= T_{total}$

Fig. 2. Complete Algorithm for Deterministic Task Graph Analysis
(The basic model can be derived by setting $D_i \equiv 0$ in the highlighted steps.)

The Complete Model Including Overhead Costs

Many programs (including three of the applications we study in Section 4) make significant use of shared resources such as the communication network or software locks, and the model must account for the cost of this resource usage. Furthermore, when resource contention is significant, these costs will depend on the number of processors and/or the specific set of tasks that are executing concurrently, and this dependence must be accounted for in the model. In deterministic task graph analysis, the mean overheads are computed by a lower level (generally stochastic) system model, and are represented as deterministic quantities that are added to the fixed CPU demand in the higher level model. Thus, the task execution sequence in the model is still unique. Therefore, other than computing and incorporating the mean overhead costs, the same four steps outlined in section 3.2 can be used to compute the overall execution time of the program. The four changes to the algorithm to compute and incorporate the mean overhead costs are highlighted with boxes in Figure 2.

In general, the mean communication overhead for each executing task must be computed for each combination of tasks in execution, i.e., for each possible set $\mathcal{E}$. Step 1 in the figure invokes a system-level model to compute the mean communication overhead for all executing tasks in $\mathcal{E}$. The overhead, $D_i(\mathcal{E}, \{M_{i,j}\}, \{r_i : i \in \mathcal{E}\})$, represents the total overhead incurred by task i *if the task had run to completion while all other tasks in $\mathcal{E}$ were in execution.* Unlike stochastic models, the number of sets $\mathcal{E}$ in an evaluation of the deterministic model is at most N (because the set $\mathcal{E}$ is fixed between task completion instants, and there are at most N such instants in the unique execution sequence). The calculation of D_i is described in Section 3.3.

Given $D_i, \forall i \in \mathcal{E}$, the total remaining execution time of each task i, assuming no state changes, is simply $r_i + D_i$. The time until the next task completion instant, T_{elapse}, is the minimum remaining task execution time over all executing tasks. This is the second modification to the basic algorithm.

Since the overheads can change in each new state, it is important to recompute the remaining *fixed* CPU demand at each completion instant, for each task that does not complete. We assume that the CPU demand for each task i diminishes at a constant rate for each state of the program, so that the rate in the current state is given by $\frac{r_i}{r_i+D_i}$. Then, for each such task i, an additional CPU time of $T_{elapse} \times \frac{r_i}{r_i+D_i}$ is completed in the interval T_{elapse}. We subtract this product from the previous value of r_i. This is the final modification to the basic algorithm.

This equation, as well as the use of the input resource usage parameters for all solutions of the lower level model, assumes that resource request rate and service time parameters are fixed throughout the lifetime of a task. "One-time" overheads, such as the delay to obtain a lock on a task queue and retrieve a task, are not subject to this assumption. Such delays are computed only once for each task i, in the state when task i is added to $\mathcal{E}$, and the delays are included in r_i. For our experiments, we modeled such a lock as an M/M/1//K queue with K set to the mean number of non-idle processes since the start of execution, and the arrival rate set to the mean interval between task completions since the start of execution.

Finally, a desirable and potentially useful property of deterministic task graph analysis is that it usually gives exactly the same results for the condensed task graph

(Table I) as for the original task graph, in cases where a condensed graph can be constructed. This is straightforward to see in the case of the basic model, and is true for the complete model if the tasks that are condensed into a single node have identical resource usage parameters. (This assumes that lock overheads, which are incurred once per task, are either negligible or that lock contention is small so that the total lock overhead can be estimated once at the beginning of each condensed task.) Our experiments with statically scheduled programs have corroborated this claim. This property could be useful because, when the condensed graph can be computed, using it would make the model solution significantly more efficient for programs with very large task graphs.

Solution Complexity of The Basic and Complete Models

The measured solution times for the basic and full model for several realistic programs are presented in Section 4. Here, we focus on their computational complexity.

For a graph with N nodes, E edges, and maximum parallelism $P_{\max}$, and with specific assumptions about the scheduling function discussed below, the overall complexity of the basic algorithm is $O(NP_{\max} + E)$. ($P_{\max}$ is defined as the maximum number of active processors in any execution of the program for the input corresponding to this task graph.) This follows from the following observations. For steps 2 and 3 of the algorithm, the size of $\mathcal{E}$ is never greater than $P_{\max}$, and for step 4, each edge in the graph needs to be examined exactly once in the overall solution. The cost of step 5 depends on the cost of evaluating the scheduling function. In any case, at most $NP_{\max}$ evaluations of the function need to be made (or $N \cdot \max(P, P_{\max})$, if values of $P > P_{\max}$ were of interest), and at most N of these will successfully find a task from L to schedule. For many common scheduling functions, including the typical static and dynamic task scheduling schemes employed in most of the applications considered in this paper, the cost of each evaluation is $O(1)$. More complex functions such as some semi-static scheduling schemes may have a cost that is $O(n)$ for a ready-list containing n tasks. Nevertheless, for many such functions it can be detected in $O(1)$ time that no ready task is available for a particular free process p. Therefore, at most $O(N)$ choices from the ready-list will have a cost that is *greater* than $O(1)$, and the cost of each will be $O(P_{\max})$. This gives the same overall complexity, $O(NP_{\max} + E)$. This category includes all the scheduling functions with cost greater than $O(1)$ studied in this paper, namely those in Section 6.3. In practice, we believe that the basic model should be extremely efficient for any practical task scheduling method.

The extra solution complexity of the complete model is due to the cost of computing $D_i, i \in \mathcal{E}$ in the first highlighted step. In each iteration, this added cost includes $O(P_{\max})$ for obtaining system-model inputs and outputs corresponding to the tasks in $\mathcal{E}$, plus the cost of solving the system-level model. The latter is usually a queueing network model where each active process is represented as a customer. Thus, if the solution cost is $O(m)$ for a queueing network with m customers, the added complexity due to the additional step will be $O(P_{\max})$ per iteration and the complete model would have the same solution complexity as the basic model, i.e., $O(NP_{\max} + E)$ overall. For example, the required condition would be satisfied by using the standard Approximate Mean Value Analysis technique (which has proved highly successful for parallel system performance analysis [Vernon et al. 1988;

Willick and Eager 1990; Adve and Vernon 1994]), and assuming that the number of queueing centers and the number of iterations per MVA solution are small.

Nevertheless, a naive implementation of the system-level solution step can dominate the overall solution time of the model. In particular, reducing the number of times the system-level model must actually be solved is usually worthwhile. In [Adve 1993], we discuss techniques to minimize model solution time in practice. The most important one is memoization of the system-level model outputs to avoid solving the system-level model multiple times with the same inputs (which could happen very often otherwise).

3.3 An Example System-Level Model for the Sequent Symmetry

The appropriate choice of system-level model for a particular study depends on the system under consideration, and on the required accuracy. Therefore, unlike many previous authors [Thomasian and Bay 1986; Kapelnikov et al. 1989; Mak and Lundstrom 1990; Harzallah and Sevcik 1995; Liang and Tripathi 2000], we do not specify any particular queueing network model to be used at the system level. In modeling applications on the Sequent Symmetry multiprocessor, we used a system-specific queueing model for the bus and shared memory system (based largely on a model presented in [Tsuei and Vernon 1992]) to calculate communication overhead costs fairly precisely. To validate our model, we compared the model predictions to direct hardware measurements of remote request latencies for these applications, and found that the model predicted mean response times within 10% of the measured values in most cases, and within 18% in all cases tested. This queueing network model is briefly described here.

The Sequent Symmetry bus uses an invalidation-based snooping cache protocol. The possible types of remote communication requests on the bus are *read* (r), $read + write_back$ (rwb) and *invalidate* (inv). For either type of read request, the required cache line is supplied either by main memory or by a remote processor's cache. Thus, we use the following parameters (assumed to be the same for each active processor) to characterize remote communication behavior on the Sequent:

- λ_{bus} Mean request rate to bus per active processor
- f_{inv} Fraction of requests that are of type *invalidate*
- f_{rwb} Fraction of read requests that are of type $read + write_back$
- p_{cache} Probability that a read request is served by a remote cache

The values of these four parameters are specified for each task i as the resource usage inputs $\{M_{i,j}\}$ to the system-level model.

Our system-level model (shown in Figure 3) is a closed single-chain queueing network model with $P_{active} = |\mathcal{E}|$ customers, the bus and the two memory modules represented as queueing centers with deterministic service times, and the caches and processors represented as infinite-server (delay) centers.[2] We use a single-chain model to avoid having P_{active} classes of customers, in order to permit solutions with complexity of $O(|\mathcal{E}|)$. Using a single-chain model requires common (average)

[2] In a single-chain network, all P_{active} customers have identical visit ratios and service time statistics. The processor and caches are single-class queues. The three types of bus requests and two types of responses are modeled as separate customer classes at the bus queue, and read and write memory requests use two separate customer classes at each memory queue [Lazowska et al. 1984].

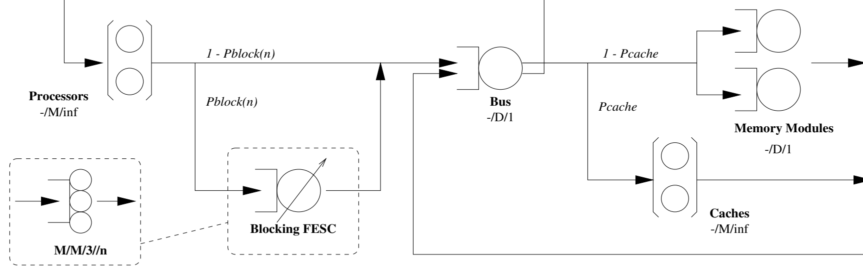


Fig. 3. Queueing network model for Sequent Symmetry multiprocessor system

values of λ_{bus}, f_{inv}, f_{rwb} and p_{cache} for all the customers. We compute λ_{bus} as the average of the values of all active tasks, i.e., $\lambda_{bus} = \Sigma_{i \in \mathcal{E}} \lambda_{bus}(i)/P_{active}$, and the other three as weighted averages, weighted by $\lambda_{bus}(i)$. For example, $f_{inv} = \Sigma_{i \in \mathcal{E}}(f_{inv}(i) \times \lambda_{bus}(i))/\lambda_{bus}$.

The equations for the response times and queue lengths at these queueing centers are described in [Tsuei and Vernon 1992]. The equations use heuristic approximations to capture two protocol features, namely, that read responses must be returned in the order that the original requests were issued on the bus, and that read responses have non-preemptive priority over processor requests.

We make a key simplification in the model to capture a third key feature of the bus protocol, viz., at most three read requests can be outstanding at any time from all processors, with at most one per processor. Tsuei and Vernon used a separate Markov chain to model the different combinations of requests that could be outstanding, and solve the above queueing network model for each state to obtain the state transition rates of the Markov Chain. Instead, we directly capture this behavior in the queueing network model by using a flow-equivalent service center (FESC) [Lazowska et al. 1984] to capture the average blocking times due to this protocol feature, and we need to solve the queueing network only once. The FESC is an additional delay center in the queueing network whose service time is a function of the customer population (see Figure 3). For $n \geq 4$, read requests visit the delay center with probability $p_{block}(n)$ *before* using the bus, and the mean delay time per visit to the delay center is $R_{block}(n)$. The parameters $p_{block}(n)$ and $R_{block}(n)$ of the FESC are estimated by solving a separate M/M/3//n queue [Gross and Harris 1985] for $4 \leq n \leq P_{active}$, *during* the overall network solution. The mean service time in this queue is set equal to the total mean residence time of a read request from the time it is transmitted across the bus until the time the response is received at the processor *at customer population* $n-1$, $\forall n \geq 4$. These residence times are intermediate results of the Mean Value Analysis solution described below.

We used customized Mean Value Analysis to solve the queueing network, which gives fairly accurate mean response times and other metrics (typical errors are less than about 10–20%) even with the approximations above [Vernon et al. 1988; Willick and Eager 1990; Adve and Vernon 1994]. Because we need the residence times of read requests $\forall n \geq 3$ (used as inputs to solve the FESC), we used the exact MVA solution algorithm which is a recursion on the customer population n from $n = 1$ up to $n = P_{active}$, instead of more common iterative approximations

such as Bard-Schweitzer [Lazowska et al. 1984]. The M/M/3//n queue is then solved once at the start of each step of this recursion, using the residence time computed in the previous step. This is fast but grows as $O(n)$, introducing an $O(|\mathcal{E}|^2)$ term in the complexity of the overall queueing network solution with $|\mathcal{E}|$ customers. This contribution was negligible for our target system since $|\mathcal{E}| < 20$. A more approximate queueing network could be used for large systems, to ensure an $O(|\mathcal{E}|)$ solution, in order to preserve the $O(NP_{\max} + E)$ complexity of the overall deterministic model.

The solution of the overall queueing network gives the average response time for bus requests, R. Then, we calculate the delay for each executing task $i \in \mathcal{E}$ as $D_i = r_i \times \lambda_{bus}(i) \times R$. Thus, a single solution of the system-level queueing network model yields D_i for all $i \in \mathcal{E}$.

3.4 Deriving and Specifying Model Inputs in Practice

The issues that arise in deriving the model inputs for a real program in practice are discussed briefly below and described in more detail in [Adve 1993]. Manually constructing a task graph and identifying the scheduling for a program is generally quite straightforward with a basic understanding of the parallelism, synchronization structure, and work scheduling in the program. This typically took a few hours to a day for realistic, moderate-size programs not written by us, including the most complex one, `Polyroots`. (Furthermore, this exercise itself can provide valuable insights into parallel program structure and performance issues.) The task CPU demands can usually be measured using software timers. Estimating resource usage parameters like cache misses accurately can be more challenging, but they can be estimated approximately using on-chip counters that are becoming available on many general-purpose processors [Browne et al. 2000]. For the validations in this paper, we directly measured the communication parameters using a hardware bus monitor, in order to minimize this as a source of error and permit precise evaluation of model accuracy.

Some research compilers also construct task graphs or equivalent information automatically for different types of parallel programs, and such support could be used both to extract task graphs and to automate the process of measuring the numerical model input parameters [Browne et al. 1995; Adve and Sakellariou 2000].

The deterministic task graph model can also be used to predict the effect of some program or system changes on program performance *before implementing the changes in the code*, simply by modifying the task graph (for certain algorithmic changes), task scheduling function, or the lower-level system model. We show three examples of such studies in Section 6. Some program or system changes, however, may affect the numerical task time and resource usage parameters. Estimating the change in task CPU times approximately is often not difficult, but communication parameters (e.g., cache misses) can be more challenging to predict. This limitation (which is shared by previous detailed models for parallel program performance prediction) is discussed further in the Conclusions.

4. EVALUATION OF THE DETERMINISTIC TASK GRAPH MODEL

In this section, we evaluate the efficiency and accuracy of the deterministic task graph model for several realistic applications. For this study, we use five shared-

memory programs on a Sequent Symmetry multiprocessor. The principal common features of the applications are that all five are scientific and engineering applications written for shared-memory systems, they are written to spawn one process per processor during execution, and they do not have significant I/O requirements. All five were written by others, either as benchmarks or for real use. The task graphs for the applications are shown in Figure 1 and the chief characteristics of the applications relevant to this study are listed in Table III. These are discussed in more detail along with the results for each application in Section 4.3.

Two of the applications that we study (`DynProg` and `Polyroots`) have negligible communication costs and the basic deterministic model suffices for these programs, except if lock contention were high for `Polyroots`. The other three applications (`MP3D`, `PSIM` and `LocusRoute`) have significant communication overhead and we compare the accuracy of the basic and full models for these programs.

4.1 Methodology Used in the Study

Our experiments were conducted on a 20-processor Sequent Symmetry S-81. We manually wrote scripts to generate each of the task graphs for given input parameters, based on an understanding of each program structure. The task CPU requirements were measured accurately using microsecond timers provided on the Sequent Symmetry. To minimize bus contention during these measurements, they were made while executing stand-alone on 1 processor. For the three applications that have significant communication overhead, the mean communication costs on one processor predicted by the system model were subtracted from the measured CPU requirements so that these costs are not counted twice in the model predictions. (This would not be required if actual CPU demands are estimated analytically rather than measured, as described in Section 3.) The specific scheduling functions used in each program are explained along with the results for the individual programs.

In measuring the resource usage parameters (listed in Section 3.3), we assumed the parameter values were identical for all the tasks in a given program phase. The average values for each phase were measured directly in hardware to permit precise model validations. A more difficult issue is how these parameter values vary when the program is executed on different numbers of processors. For the validations on a small system such as the Sequent (with only 20 processors), we assumed that the behavior would stay approximately constant for 2 or more processors, and that the behavior on 1 processor could be substantially different. We tested this assumption for the mean communication rate in `MP3D` and `PSIM`, and found it to be approximately true for both programs.[3] Alternatively, either separate measurements or a separate analytical model would have to be used to derive the appropriate resource usage parameters for different numbers of processors, as discussed in Section 3.4.

The accuracy of model predictions were tested by comparing against actual measured program execution times for each program, measured separately on different

[3]Specifically, we measured the mean bus inter-request time in a phase, for different numbers of processors. For the dominant phase of `MP3D`, for example, the values we obtained on 1, 2, 4, 6, 8, 10, 12 and 16 processors were 539, 443, 404, 418, 415, 429, 447 and 452 bus cycles respectively. Similarly, for the larger phase of `PSIM`, the values obtained on 1, 4, 8 and 16 processors were 79, 75, 72 and 78 respectively.

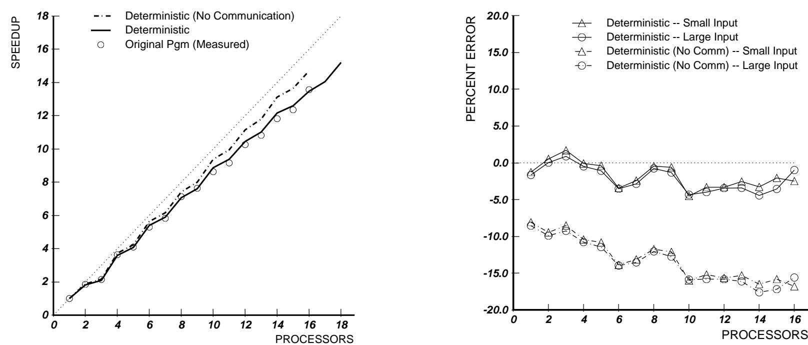
Table III. Applications Evaluated using the Analytical Models

Name	Description	Task Graph	Task Scheduling	Perf. Losses
MP3D (C; 1858 lines)	Particle simulation in rarefied fluid flow	*Fork-join*: five parallel loops per iteration; one loop has more than 90	Static allocation of loop iterations in each loop.	Cache misses; Small load imbalances
PSIM (PCP; 2495 lines)	Multistage interconnection network simulation	*Fork-join*: two parallel phases per iteration (6 parallel loops per phase with widely differing granularities); barrier at the end of each phase	Static allocation of loop iterations; processes "split" between different parallel loops	Cache misses; load imbalances due to process splitting
Locus Route (C; 7199 lines)	Wire routing in VLSI standard cells (Commercial quality)	*Fork-join*: two parallel phases; widely varying task sizes per phase	Dynamic allocation with single FIFO task queue in each phase	Cache misses; load imbalance due to task skew
Polyroots (C; 3396 lines)	Compute roots of a polynomial with arbitrary precision (integer) coeffs.	*Non-series-parallel*: unstructured graph with widely varying task sizes	Dynamic allocation with single FIFO task queue	Load imbalances; limited overall parallelism
DynProg (C; 2691 lines)	Aligning 2 gene sequences via dynamic programming	*Non-series-parallel*: pipelined array of numerous small but uniform tasks;	Static round-robin allocation of rows of tasks to processes	Limited parallelism at start and end

numbers of processors. All measurements above were made when no other user programs were actively using the system.

4.2 Solution Efficiency of the Model

For the five programs studied in this paper, for the larger of the two inputs listed for each code in Figure 1, the task graphs ranged in size from 348 to 40963 tasks. We used the basic deterministic model for two of the programs, `DynProg` and `Polyroots`, and the full model for the other three programs. For many combinations of programs and input sizes, the model solution is virtually instantaneous. Overall, the two most expensive programs to analyze were `PSIM` and `DynProg`, where the task graph with the larger input had 40963 and 32003 tasks respectively. For the former, which is a fork-join program, the model could be solved in under 9 seconds on a DECstation 5000/125, and required less than 2.6 megabytes of memory. The latter case required the longest solution time (30 seconds) and the largest memory capacity (about 6 megabytes) of all the results presented in this paper. Even though `PSIM` had the larger task graph, it was less expensive than `DynProg` because its graphs are significantly simpler. In particular, the graphs of `PSIM` contain large groups of tasks with similar behavior that can be manipulated more efficiently, because of simple optimizations in the implementation [Adve 1993]. Overall, we found that the deterministic task graph model is quite efficient for programs with moderately large and complex task graphs.



(a) Speedups for large input ($T(1) = 9.9$ sec) (b) Error in predicted execution time

Fig. 4. Predicted and measured performance of PSIM

4.3 Accuracy of the Model

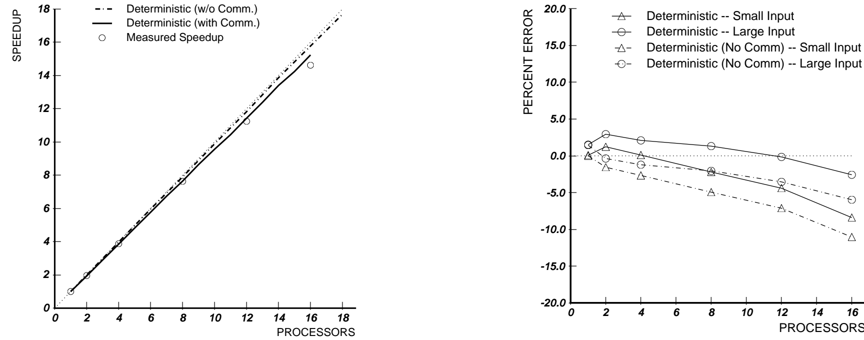
Results for PSIM

PSIM is an interconnection network simulator from Lawrence Livermore Laboratory, written in PCP, a parallel extension of C that supports efficient nested forking within programs [Brooks III 1988]. It is a fork-join program with two parallel phases per iteration, with each phase effectively ending in a barrier. Each parallel phase consists of 6 parallel loops with no intervening barriers (Figure 1(b)). The tasks correspond to individual loop iterations. The tasks of each loop are statically allocated in cyclic order *to the processors that execute the loop.* Two of the six loops are executed by all processors. Of the other four loops (two pairs), one pair of loops is executed only by the even numbered processors while the other pair is executed only by the odd numbered processors (unless, of course, only a single processor is being used). This static scheduling policy can be concisely described in our scheduling framework mentioned in Section 3.1. The work per task (loop iteration) is much larger in some loops than in others.[4] The processor-splitting between loops is done in a way that ensures the load is well balanced when P is *even*, but significant load imbalance occurs when P is *odd*.

We consider two input networks for PSIM, containing 1024 and 4096 processors respectively. Figure 4 shows the percentage errors in predicted execution time for the two input sizes from the basic and full models, and the measured and predicted speedups for the larger input (including the predicted speedup ignoring communication overhead, using the basic model). The predicted speedups are relative to the predicted execution time on $P = 1$. The caption shows the measured execution time on $P = 1$ (denoted $T(1)$).

The full deterministic task graph model is consistently accurate for this program, yielding execution time predictions within 4% of the actual measured values. Thus, even the widely varying task times of PSIM are accurately represented (along

[4]In fact, for a particular input, we observed that the mean task times in the six parallel loops of the second phase were 44, 677, 7, 473, 7 and 42 microseconds respectively, with very little variation around the mean within each loop.



(a) Speedups for large input ($T(1) = 155$ sec.) (b) Error in predicted execution time

Fig. 5. Predicted and measured performance of MP3D

with mean communication costs) by a set of deterministic values. Furthermore, the model precisely tracks the variations in execution time between odd and even numbers of processors, because the unusual, non-uniform, task scheduling method used in the application is precisely and directly represented in the deterministic model.

This application also has substantial communication overhead due to frequent cache misses and bus contention. This is reflected in the higher errors for the basic model (Figure 4 (b)), which ignores communication overhead but is otherwise identical to the full model. (In fact, we measured bus utilization to be 0.81 for the larger input size on 16 processors.)[5] The basic model is clearly insufficient for this program. The full model is consistently accurate for this code, showing that it accurately captures the communication costs including contention. Furthermore, despite the communication and contention, ignoring the variance in communication delays and task times again appears to have little impact on model accuracy. (Although these results are obtained on a system with relatively low memory latencies, our previous work has shown that even with much higher and more variable latencies, we do not observe significant variability in execution time between synchronization points in shared memory programs [Adve and Vernon 1993]. We therefore expect the model to continue to be accurate on such systems as well.)

Results for MP3D

The next program, MP3D, is taken from the SPLASH suite of parallel applications [Singh et al. 1992], from Stanford University. It simulates the motion of particles in very low density fluids. The task graph for one iteration of this program is shown in Figure 1(a). It is a fork-join task graph with five parallel phases (parallel loops). In each parallel loop, chunks of 8 consecutive loop indices are always allocated as a single unit and hence we can consider a chunk to be a single task (see Table I).

[5]The processors on the Sequent Symmetry are extremely slow relative to bus and memory speed, so that the relative impact of the communication overhead (both latency and contention) is much smaller than would be expected on systems with relatively faster processors. For example, the entire round-trip latency for a read request from any processor to main memory is only 8 cycles, of which the bus itself is occupied for only 3 cycles in all.

There are small but significant variations in the task times in each parallel loop. The tasks of each loop are statically allocated to the processes in cyclic order.

We consider two input sizes for `MP3D`: a small input size of 5000 particles, and a somewhat more realistic input size of 20000 particles. Figure 5 shows the results, and has the same organization as the figure for `PSIM`.

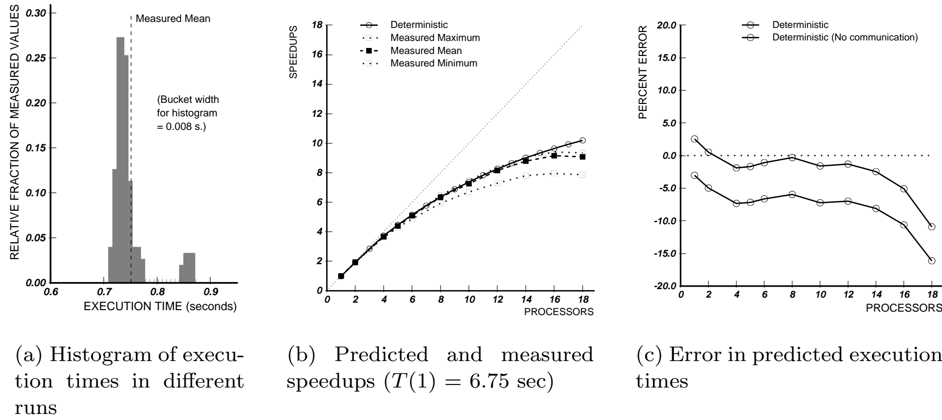
The results show that the deterministic task graph model is highly accurate for `MP3D`, as well. This accuracy is possible because the variations among the task times in each parallel loop are represented precisely by using a set of deterministic values, and the task scheduling is also represented precisely and directly in the model. The figure also shows that the mean communication overhead costs are small but measurable for this application, and are captured precisely by the system-level model. Only the variance in the individual task times, which arises in this program primarily due to communication delays, is ignored, and the results indicate that ignoring this variance had little impact on the accuracy of the results. Either the basic model or the full model could be useful for studying this program, depending on the desired accuracy of the modeling study (but note the communication overheads are likely to be much higher on a system with relatively faster processors).

The error in the model increases (becomes more negative) slowly with P because some small serial portions of the program were ignored when constructing the task graph. Such factors could be included for greater accuracy, but this may not be necessary even on larger systems if proportionally larger input sizes are used.

Results for `LocusRoute`

`LocusRoute`, also a SPLASH application [Singh et al. 1992], is a commercial-quality wire-router for VLSI standard cells. It is a fork-join program with two iterations, each ending in a barrier (Figure 1(c)). The computation is organized as tasks with one of three levels of task granularity, chosen by the user. We examine the coarsest granularity, namely one task per wire, because finer levels of granularity have poor performance on this system size. Modeling a finer level of granularity would require a larger task graph but with otherwise the same structure, and would not be significantly more difficult. The task CPU requirements vary widely within each iteration. For example, for the input circuit `bnrE.grin`, most tasks require less than 10 milliseconds of execution time while a few require 100 milliseconds or more. Two scheduling methods that are orthogonal to the choice of task granularity are also available in the program. For our validation experiments, we used dynamic task scheduling from a single FIFO task queue. The other choice is a semi-static scheduling method, and we study this and variants of it using our model in Section 5.3.

The CPU requirements of the tasks in `LocusRoute` can vary slightly from one execution to the next [Adve 1993] because the computation for each task depends on the order of completion of previous tasks in the same iteration [Singh et al. 1992]. This can cause the overall execution time to vary significantly from run to run, for the same input, an effect that can be magnified by dynamic task scheduling with unequal tasks. Figure 6 (a) shows a histogram of the measured execution times in 150 runs of `LocusRoute` on 16 processors, for the input circuit `bnrE.grin`. To provide a complete picture of model accuracy for this program, we compare model



(a) Histogram of execution times in different runs

(b) Predicted and measured speedups ($T(1) = 6.75$ sec)

(c) Error in predicted execution times

Fig. 6. Predicted and measured performance of `LocusRoute` (Input circuit: `bnrE.grin`)

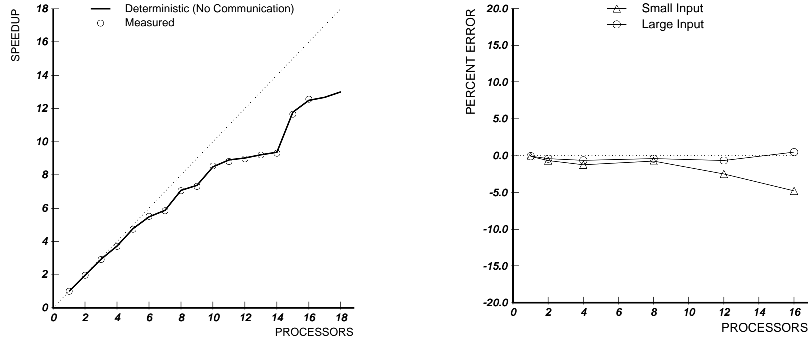
predictions against the *range* of measured execution times, i.e., the minimum and maximum, as well as the mean.[6]

All speedup values are computed relative to the *mean measured* execution time on 1 processor. The range and mean of the measured values in the following experiments were obtained from 40 runs for each number of processors.

Figure 6 (b) compares the speedup predicted by the full deterministic model with the range and the mean of the measured speedups. Figure 6 (c) shows the error in the predicted execution times for both the basic and the full model, relative to the mean measured execution times. The full deterministic task graph model is quite accurate for this program compared to the mean measured values. The basic model has significantly higher errors, showing that communication overhead *is* significant, and in particular that it is important to use the full model for understanding the performance of this program. An interesting feature of this program is that highly non-uniform task times combine with the order of execution of tasks to introduce a large load imbalance in the program. Specifically, in this input circuit, an unusually large task appears towards the end of the queue in each iteration and thus a significant part of each iteration is spent executing this task alone (this effect is discussed further in Section 5). The accuracy of the full deterministic model demonstrates that the model has successfully captured the impact of the *order* of execution of tasks, in addition to the precise allocation of tasks to processes.

Figure 6 (b,c) also show that the error relative to either the mean or the most distant measured value shows an increasing trend at higher values of P. Additional detailed measurements showed that the individual task CPU requirements (excluding communication costs) on 16 processors were significantly higher than the corresponding values measured on 1 processor, due to the non-deterministic nature of the task CPU requirements. Since the 1-processor values are used as inputs to

[6]The individual task times were measured using only a single run of the application and used as input to the model, i.e., the model predicts the performance for one possible realization of the task times. This is more appropriate than averaging individual task times across multiple runs since the variability of different tasks may be correlated.



(a) Speedups for large input ($T(1) = 398$ sec.) (b) Error in predicted execution time

Fig. 7. Predicted and measured performance of `Polyroots`

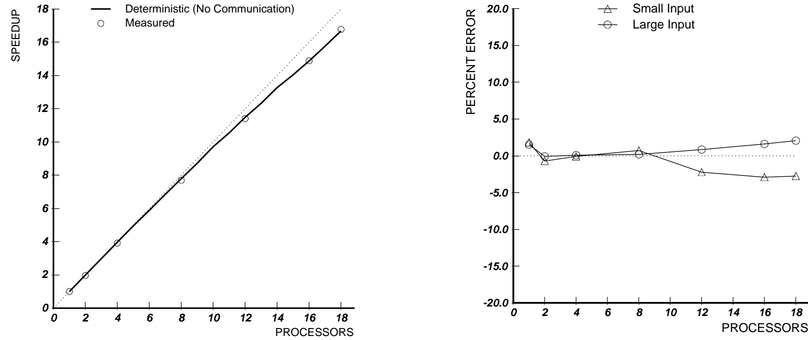
the model, the predicted execution times for $P = 16$ are low. Similar measurements also showed that this effect is much less significant on $P = 2$ and $P = 8$, matching the observed trend in the model prediction errors. Overall, despite the variability of task CPU times across runs, our model predictions appear sufficiently accurate (both qualitatively and quantitatively) for exploring program performance issues.

Results for `Polyroots`

The remaining two programs, `Polyroots` and `DynProg`, have non-series-parallel task graphs. `Polyroots` computes the roots of a polynomial with arbitrary-precision integer coefficients [Narendran and Tiwari 1992]. The task graph of this program is fixed for a particular input polynomial degree, and is shown in Figure 1(d) for a polynomial of degree 20. The tasks of `Polyroots` have very widely varying execution times both within and across task groups (shown as boxes in the figure), with the largest tasks at the leaves. The tasks are dynamically scheduled using a single task queue, as in `LocusRoute`.

Figure 7 shows that the model is remarkably accurate for this program. The percentage errors in the predicted execution times for the larger input are all *less than 1%*! With the smaller input size, the errors are slightly higher because the program has some small forking and communication overheads (which were ignored in the model) and these are relatively more significant with the smaller input size and greater parallelism. The measured speedup (shown in Figure 7 (a) for the larger input) increases non-uniformly with P because of the changing allocation of tasks that have widely varying task times. The predicted speedups accurately track the measured speedups since the model precisely represents the allocation of individual tasks to processors *and* the order of task execution.

The above experiments with `Polyroots` as well as `LocusRoute` ignored the overhead for accessing the critical section (lock) protecting the shared task queue. It is interesting to evaluate how accurately the full deterministic task graph model represents contention for such a shared software resource since the number of lock accesses between synchronization points may be relatively small (e.g., much smaller than the number of communication delays), potentially introducing more significant variance into process execution times. We inserted an artificial exponentially



(a) Speedups for large input ($T(1) = 11.7$ sec.) (b) Error in predicted execution time

Fig. 8. Predicted and measured performance of `DynProg`

distributed delay into the lock holding times in `Polyroots`. We modeled the lock as an M/M/1//K queue as described in Section 3.2. We compared the model predictions to the measured execution times for mean lock holding times of 1, 10 and 50 milliseconds. The data obtained (omitted for lack of space) show that despite significant contention for the lock causing much higher total execution times, the model errors are again small, and are higher than 5% only when lock holding times are very large (10 milliseconds or more) and total processing time is small (the smaller input).

Results for `DynProg`

`DynProg`, uses a pipelined dynamic programming algorithm for aligning two gene sequences [Lewandowski et al. 1996]. The program has a pipelined task graph (Figure 1(e)) with $O(G^2)$ tasks for an input containing two gene sequences of size G each. The tasks are quite uniform in computational costs, and are of much smaller granularity than many of the tasks in `Polyroots`. All the tasks in a row of the main task array within the graph are allocated to the same processor; rows are statically allocated to processors in round-robin fashion. Our framework enables this scheduling method to be specified simply by using a separate task queue per processor with no switching of queues, and using a user-enumerated initial allocation of tasks to each queue.

For `DynProg`, we again used two input sizes, namely G=100 and G=500. The percentage errors in the predicted execution time for the two input sizes are shown in Figure 8 (b) and the predicted and measured speedups for the larger input are shown in Figure 8 (a). Even though the absolute execution times are about two orders of magnitude smaller than `Polyroots`, the errors are still within the range of 1-3% in all cases. These results again indicate that the basic deterministic model is extremely accurate for programs to which it applies. The results also demonstrate that the model can be used for large and fairly complex task graphs.

5. EXAMPLE APPLICATIONS OF THE DETERMINISTIC TASK GRAPH MODEL

We claimed earlier that the deterministic task graph model can be useful for evaluating program design issues. In this section, we illustrate this point by evaluat-

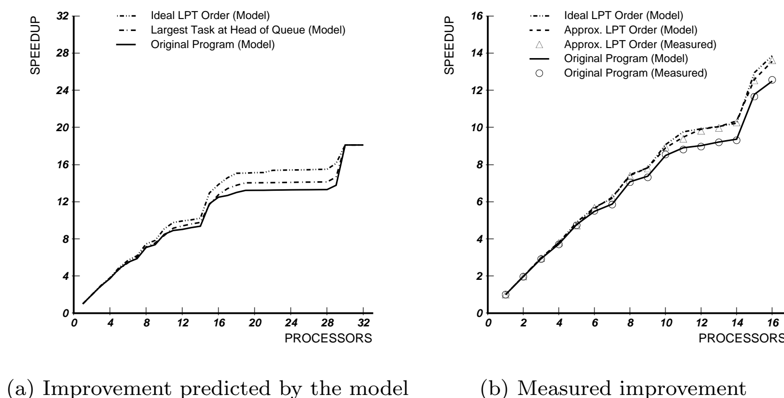
ing several such issues for `Polyroots`, `PSIM` and `LocusRoute`. For each of these programs, insight obtained by applying the model led to one or more suggested program design changes, each of which could be evaluated *a priori* (i.e., prior to implementing the changes) using the model. For two of the programs, the performance improvement predicted by the model is also shown to be accurate by implementing the changes and measuring their performance.

5.1 Evaluating Design Choices for `Polyroots`

The speedup curves for `Polyroots` in Figure 7 (a) show that the speedup of this program is substantially less than linear, is not smooth, and can be particularly poor at some values of P. To aid in determining the source of the poor performance, we examined detailed time-lines of the execution of tasks by each process, computed from intermediate results of our model (Figure 7.2 in [Adve 1993]). This data provided three basic insights: (1) early phases of the program have insufficient and varying parallelism; this is inherent in the algorithm, (2) the final phase, a parallel loop with $P_{\max}$ tasks, requires more than half the total execution time with $P = 24$ (for example), and has significant load imbalance due to a few "leftover" tasks when $P_{\max}/P$ is not an integer; this accounts for the non-smooth speedup, and (3) the two largest tasks in the phase (the last task executed by processes 18 and 23) are among the last to begin execution, thus exacerbating the load-imbalance. This discussion illustrates that our analytical model can provide detailed quantitative insight into program behavior, comparable to what measurement or simulation tools can provide, at much lower cost. This is a key goal of our work, as noted in the Introduction.

Observations (2) and (3) above immediately suggest that one simple improvement would be to place the largest task, which is trivial to identify, at the head of the task queue. Furthermore, if all task processing times in the phase are somehow known, better performance might be obtained by scheduling the tasks in decreasing order of execution time, a heuristic called the LPT (Longest Processing Time) rule [Horowitz and Sahni 1984]. Many programmers have enough insight into the task behavior of their programs to implement an *approximate* version of this rule by using simple estimates of the task times to order the tasks, particularly when there are wide disparities in task times. For example, the LPT order can be approximated in `Polyroots` using very little additional computation at the start of the final phase, simply by sorting the intervals of the real line in order of decreasing length. We call this the Approximate LPT order.

No general-purpose model can predict the performance of the Approximate LPT order (before it is implemented in the program) because the estimation of the order is algorithm-dependent. Instead, the model can predict the performance of the "Ideal LPT" order that would result if the actual task times of the original program were used to reorder the tasks (these are available assuming the model has already been applied to identify performance problems in the original program). For our model, we can specify the changes in order simply by reordering the task CPU requirements in the model input. The predicted speedups with the two heuristics (Largest Task First and Ideal LPT) are compared to the original program in Figure 9(a). The graph shows that merely moving the largest task to the head of the queue would yield a small though useful improvement for $P \leq 16$, but executing



(a) Improvement predicted by the model (b) Measured improvement

Fig. 9. Program `Polyroots`: the effect of reordering tasks in final Phase

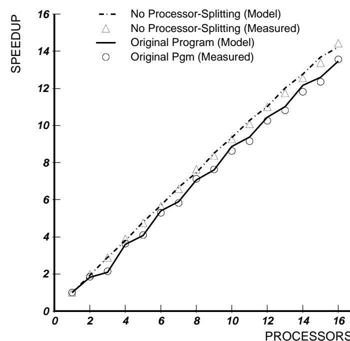


Fig. 10. Improvement in speedup of `PSIM` without splitting processors between loops

the tasks in the Ideal LPT order would yield a more significant improvement in speedup over a wide range of P.

The model results indicate that it could be worthwhile to implement the LPT heuristic in the program, assuming the approximate order will come reasonably close to Ideal LPT in performance. We implemented the change and measured its performance. Figure 9(b) compares the measured speedups of this modified program to those of the original program and to the predicted speedups for Ideal LPT. (As an extra validation exercise, we also recorded the new order of tasks in the modified program, allowing the model to predict the performance of Approximate LPT *ex post facto*. The figure shows that these predictions are again very accurate compared with measurements.) Most importantly, the measured results show that the simple approximation to Ideal LPT was able to realize almost the full improvement predicted for the ideal version.

To summarize, the model provided insight into a key performance bottleneck for this program, predicted the potential performance impact of two possible modifications (using an idealized version of one of them), and correctly predicted that

it would be worthwhile to attempt the more aggressive modification, namely, a full reordering of tasks. These predictions were possible because of the accurate representation of task scheduling and the accurate prediction of synchronization costs.

5.2 Evaluating A Possible Change to PSIM

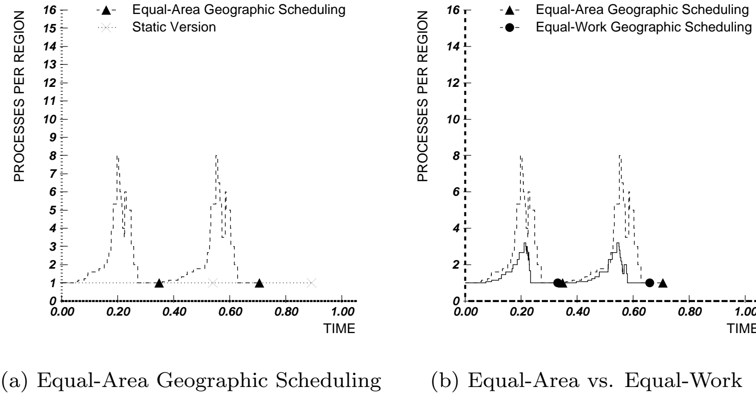
We next look at the program PSIM, focusing on the effect of the processor-splitting scheduling method, which allocates different loops to even and odd numbered processors in each phase. The results in Sections 4 and 5 showed that this scheduling method produces measurable load-imbalance as well as non-smooth speedup behavior. The deterministic task graph model can be used to predict the speedup that would be obtained if, instead, the iterations of each loop are statically scheduled across all processors, as in MP3D. The model predicts that this change would give a 5-10% improvement in speedup as well as a smooth speedup curve, as shown in Figure 10. We also implemented this change and measured its performance. The figure shows that the model predictions were again very accurate, and the improved program achieves essentially the improvement that was predicted. We also used the model to examine dynamic scheduling of the loop iterations (ignoring scheduling overhead). In this case the predicted further improvement was negligible, and we did not implement this.

5.3 Evaluating Communication Locality and Load Balancing in LocusRoute

In some applications, the choice of task scheduling method introduces a trade-off between data locality and load-balancing, and studying this trade-off analytically is a challenging problem. In LocusRoute, for example, the principal communication arises when two or more processes route wires through overlapping regions of the VLSI chip [Singh et al. 1992]. To reduce this communication, LocusRoute provides a semi-static task scheduling option called *geographic scheduling* in which the chip is divided into a number of regions of equal area and a separate task queue is maintained for each region. Each process initially works on a separate task queue to minimize communication, but is re-assigned to another queue when its current queue becomes empty (choosing the one with the fewest number of other processes). Thus, the need for dynamic load-balancing can compromise the locality of communication.

The actual data locality in LocusRoute is complex and difficult to predict. To gain some intuition about the trade-off between data locality and load balancing, we use the model to compute the average number of *active processes per active task queue* as a function of time during the predicted execution sequence for the program tasks. This metric is not specific to LocusRoute, and this and other similar metrics can be automatically generated for the scheduling framework used in the model. For LocusRoute, this metric gives a qualitative view of data locality because a lower value indicates that relatively fewer processes are sharing and updating shared data for each chip region (similar interpretations should be possible for other codes that assign tasks to queues based on locality).

Figure 11(a) plots this average as a function of time in an execution on 16 processors, using 16 regions for geographic scheduling. We also use the model to plot a hypothetical static version of geographic scheduling that has the same initial al-



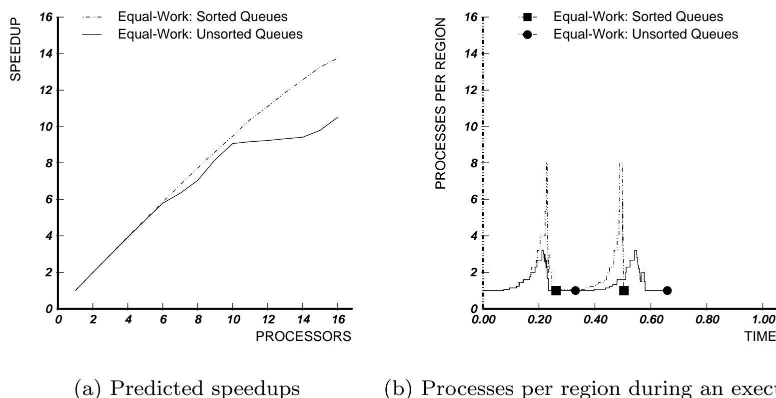
(a) Equal-Area Geographic Scheduling (b) Equal-Area vs. Equal-Work

Fig. 11. Mean number of processes per region during an execution of `LocusRoute` ($P = 16$)

location as the original, but in which a process does not switch task queues after its own queue becomes empty. The pair of points on each curve mark the predicted end of the first and second iterations of the program. The curve for the static version is constant at 1, but each iteration lasts much longer because of the poorer load-balance. In the actual (semi-static) geographic scheduling, however, the average changes as processes switch from empty to non-empty queues. The figure shows that (1) the initial distribution of work among the 16 regions is highly unbalanced since some processes switch regions very early, and (2) for a substantial portion of each iteration the average number of active processes per active region is quite large, which can cause significant interprocess communication. The key conclusion is that *an unbalanced initial division of work may significantly compromise data locality.*

We can achieve a more balanced initial division of work among the processors by dividing the chip into rectangular regions of approximately *equal work*, using the area of the smallest rectangle containing a wire as the measure of the work required for the wire. By specifying the new initial allocation of tasks to queues, the deterministic task graph model can again predict the evolution of the average number of active processes per region. The results are shown in Figure 11(b). The figure shows that the new "equal-work" geographic scheduling reduces the fraction of time for which the average number of processes per queue is greater than one, and also reduces the average number of processes per queue over substantial intervals of the program, indicating that data locality should be significantly improved. Although we do not know the precise impact on execution time, we conclude that the heuristic has sufficient potential to justify its implementation.

Figure 11(b) also shows that in either geographic scheduling option, a single large task is active at the end of each iteration, indicating that the LPT heuristic might benefit this program as well. Using the model (and ignoring any effect on locality for now), we find that this change has the *potential* to significantly improve speedup for $P > 10$, as shown in Figure 12(a). Analyzing the precise impact of the LPT order on locality is extremely difficult, but we can again use the model to compute the processors-per-region metric, and this is shown in Figure 12(b). As would be



(a) Predicted speedups (b) Processes per region during an execution

Fig. 12. Predicted impact of reordering tasks with balanced semi-static scheduling in `LocusRoute` ($P = 16$)

expected with the LPT heuristic, we find a higher average number of processes per queue towards the end of each iteration. The model shows, however, that for only a small portion of the iterations, the average for the LPT queues is higher than for the unsorted queues. We conclude from these results that the approximate LPT ordering worsens locality but improves load balancing significantly, and the potential improvements justify its implementation.

In the above experiments with `LocusRoute`, metrics computed by the deterministic task graph model provided insight into various task scheduling algorithms that represent different trade-offs between load-balancing and data locality. While communication parameters such as cache miss rates are difficult to obtain for scheduling policies that have not been implemented, the model can be used to provide some insight into the performance impact of program design choices that affect communication costs as well as load balancing. The experiments with `LocusRoute` as well as `Polyroots` also show that the model internally contains a great deal of detailed performance information that can be used in innovative ways to identify performance problems.

6. COMPARISON WITH RELATED WORK

The first author's dissertation [Adve 1993] provided a qualitative characterization of previous models and a quantitative evaluation of representative stochastic models to understand the state of the art (focusing on detailed, quantitative models). Here, we briefly contrast previous models with our model, and then compare the results from our evaluation of representative models with the results for our model.

6.1 Overview of Previous Work

Some of the most successful analytical models for parallel programs are simple parametric models that estimate program performance based on one or a few parameters describing the parallelism and communication in a program [Amdahl 1967; Gustafson 1988; Hack 1989; Flatt 1984; Flatt and Kennedy 1989; Eager et al. 1989; Vrsalovic et al. 1988; Cvetanovic 1987; Blumofe et al. 1995; Culler et al. 1993; Alexandrov et al. 1995; Frank et al. 1997]. Such models are extremely useful for

obtaining broad, qualitative insights and bounds on program performance, communication overheads, and scalability. However, such models generally would not be able to predict more detailed aspects of performance such as the uneven speedup of `PSIM` and `Polyroots`, the effect of task scheduling improvements such as those studied in the last Section, or the impact of execution state on communication costs and resource contention. [Adve 1993] presents experimental data comparing the insights obtained from selected parameteric models and bounds with those obtained from our model.

Our work is complementary to these simple but insightful models, and is aimed at more detailed performance analysis and prediction, based on detailed program information. For the rest of the discussion below, we focus on more detailed quantitative models that have goals similar to ours.

Models Applicable to Arbitrary Task Graphs

Thomasian and Bay [Thomasian and Bay 1986], Mohan [Mohan 1984], and Kapelnikov, Muntz and Ercegovac [Kapelnikov et al. 1989] have proposed similar two-level, hierarchical models applicable to programs with arbitrary task graphs and arbitrary task scheduling disciplines. The higher-level model in each case is a Markov chain, and the different solution algorithms they use all have time and space complexity that is exponential in the maximum parallelism of the program. Nevertheless, these are the most detailed and general stochastic models available, and we examine the expected accuracy of these models in Section 6.2.

Tsuei and Vernon [Tsuei and Vernon 1990] propose a model based on a parallelism profile rather than a detailed task graph. In practice, this approach is only practical to apply to fork-join programs with good load balance [Adve 1993], and in fact, their model is shown to be accurate for three such programs. In contrast, the results in Section 4 show that our model is consistently accurate for a much wider class of programs.

Fahringer [Fahringer and Zima 1993; Fahringer 1993] describes a collection of compiler-driven models for predicting components of program performance: computation times, load-imbalance, message-passing costs, and per-node cache performance. The models are designed for *regular* data-parallel programs and static loop scheduling, because he uses integer polyhedrons to represent iteration counts and communication volumes. Of the applications we have studied, only `DynProg` and perhaps `PSIM` could be written in this form. For such programs, the major benefit of his approach is that it is fully automatic. Mendes and Reed [Mendes and Reed 1998] use a data-parallel compiler to generate symbolic performance estimates for regular data-parallel programs with specific communication patterns (constructed by the compiler itself). Relative to our work, their approach approach has similar benefits and limitations to that of Fahringer.

Two papers [Xu et al. 1996; Jonkers et al. 1995] have developed models similar to ours, but both are much more restricted in modeling task scheduling. Xu et al. describe a graph-based model and solution technique similar to ours (based in part on our work [Adve 1993]). However, they only consider random allocation of tasks to threads, and they directly measure communication costs for a few input and system sizes and incur significant errors when extrapolating to a different numbers of processors. Jonkers et al. describe a queueing network approach that

is similar to our deterministic task graph model, but they do not explicitly model the scheduling of tasks to processes, and only fairly restricted task scheduling can be captured accurately in the resource demands of the processes. Their model is only validated for a simple matrix-multiply loop. In contrast to both these models, we can model much more realistic and complex task scheduling methods, and our model is consistently accurate for a wide range of programs.

Schopf and Berman [Schopf 1997; Schopf and Berman 2001] describe a modeling methodology (not a specific model) called structural modeling, where an application is decomposed into a few components and a high-level structural equation is developed to compose metrics from individual component models into an overall model. To model applications on shared clusters with varying loads, they show how stochastic component metrics can be represented using limited classes of stochastic values (e.g., normally distributed values or intervals) [Schopf and Berman 2001]. Although their decompositions and structural equations are conceptually similar to a task graph and its solution, they are constructed in an application-specific manner for each individual application (they present specific structural equations for several simple computational kernels such as SOR, LU, and a genetic programming algorithm). In addition, their work does not specify whether or how they could model complex applications or components, e.g., with arbitrary task graphs such as `Polyroots` or sophisticated scheduling disciplines such as the ones in `LocusRoute`.

There are several compiler-driven tools for parallel program performance evaluation, in which the performance analysis is based either on simulation [Dikaiakos et al. 1994; Dikaiakos 1994; Parashar et al. 1994] or measurement and extrapolation [Balasundaram et al. 1991]. In general, these are all focused on message-passing systems and static scheduling disciplines. There have also been measurement-based tools that use statistical model fitting to construct performance models, and these can conceivably handle arbitrary programs (e.g., [Brewer 1995; Crovella et al. 1995]). Two drawbacks, however, are the high cost of the numerous measurements required, and the lack of insight into the causes of performance problems. Finally, there are alternative approaches based on computing bounds for task graphs with known task time distributions [Hartleb and Mertsiotakis 1992; Yazici-Pekergin and Vincent 1991]. These techniques only apply when $P = P_{\max}$ (such as in the condensed task graph under static scheduling), and their accuracy is sensitive to the size of the graph and to the task graph structure.

Models for Series-Parallel Task Graphs

Mak and Lundstrom describe a polynomial-time solution technique for series-parallel task graphs with exponential task execution times [Mak and Lundstrom 1990]. [7] Their solution heuristic ignores task scheduling and processor contention in the task-level model. These must be accounted for in the queueing network model of the system, but (like [Jonkers et al. 1995]) this is only possible in practice for restricted task scheduling disciplines. They do not present validation results for real programs. We discuss the efficiency and accuracy of this model in Section 6.2.

[7] Although task times with lower variance can be modeled by a phase-type distribution with a sequence of exponential tasks, the results in Section 6.2 show that the ML model solution is too expensive to permit such an approach.

Recently, Liang and Tripathi [Liang and Tripathi 2000] have proposed a model that generalizes the techniques of Mak and Lundstrom to analyze programs with a much wider class of task graph structures based on arbitrary combinations of series, parallel, parallel-OR, and branching subgraphs (e.g., they can model `Polyroots` but not `DynProg`). Unlike most other models discussed in this paper, they also represent multiple parallel programs in their system queueing network model. They again assume exponential task time distributions for analytical tractability. Like Mak and Lundstrom, they ignore task scheduling in their task-level model and instead capture it by representing processors as queueing centers in their queueing network, assuming product-form scheduling disciplines (e.g., processor sharing or FIFO with single-class exponential service times). They validate their model only against simulations of synthetic task graphs with exponential task times, and do not present validation results for real programs. We have not studied the accuracy of this model. For a single program with a series-parallel task graph, we expect their accuracy to be similar to that of the Mak and Lundstrom model because they use similar models of program and system for this case.

van Gemund [van Gemund 2003; van Gemund 1996] has developed a modeling language (Pamela) and a tool that constructs symbolic performance estimates from program descriptions in this language. When applicable, this tool produces closed form models that could be fairly intuitive and very fast to evaluate. Because such a language uses predefined language primitives that must be used to describe parallel program structure, we believe such an approach is limited to well-structured program kernels rather than arbitrary applications. The Pamela language is restricted to series-parallel task graphs, and to simple static or work-conserving dynamic task scheduling. The implementation of the tool uses very simplistic bounds for describing computing communication costs with resource contention. To our knowledge, his tool has been applied to small algorithmic kernels (e.g., Gaussian Elimination, parallel vector sort, and others) and to synthetic series-parallel task graphs, but not complete applications [van Gemund 2003].

Finally, there have been a a few models restricted to specific non-fork-join task graph structures such as divide-and-conquer task graphs [Madala and Sinclair 1991] or pipelined task graphs [Lewandowski et al. 1996; Sundaram-Stukel and Vernon 1999]. For applications that match these structures, such models can be attractive because they are intuitive to develop and use, and (usually) also efficient and accurate.

Models Restricted to Fork-Join Programs

There a number of previous models restricted to programs with *fork-join task graphs* [Dubois and Briggs 1982; Heidelberger and Trivedi 1983; Towsley et al. 1990; Ammar et al. 1990; Kruskal and Weiss 1985; Vrsalovic et al. 1988; Cvetanovic 1987; Tsuei and Vernon 1990], Of these, perhaps the most important is the seminal model of Kruskal and Weiss [Kruskal and Weiss 1985]. They derive a simple, closed-form estimate for the total execution time of a single fork-join program phase having N independent parallel tasks with i.i.d. task execution times of mean μ and variance σ. They assume that tasks are allocated dynamically to processors from a common queue in fixed size batches of K tasks (incurring a fixed overhead of h time units). They validated the model for a number of hypothetical

task time distributions. We evaluate the accuracy of this model for real programs in Section 6.2.

More recently, Harzallah and Sevcik use a simple linear model of barrier synchronization cost (as a function of P) to analyze performance of fork-join shared memory programs [Harzallah and Sevcik 1995]. They do not model task scheduling explicitly. They analyze communication costs using a standard Mean Value Analysis framework and workload model [Willick and Eager 1990; Adve and Vernon 1994]. They also develop separate, *algorithm-specific* analyses to obtain communication parameters. Their MVA framework and their analytical parameter estimates could be valuable to combine with our high-level model as well.

6.2 Quantitative Comparisons with Deterministic Task Graph Analysis

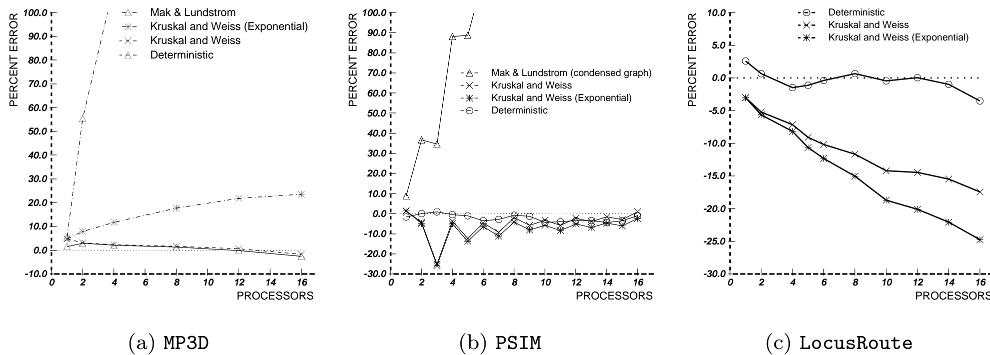
As mentioned earlier, we have evaluated several representative stochastic models for the same applications used in Sections 4 and 5 [Adve 1993]. The papers describing these models do not present evaluations using real programs. The primary goal of our comparison was to bring out the key advantages and disadvantages of assuming deterministic rather than stochastic task times.

We considered key stochastic models for fork-join [Kruskal and Weiss 1985], series-parallel [Mak and Lundstrom 1990] and general task graphs [Thomasian and Bay 1986; Mohan 1984; Kapelnikov et al. 1989]. The numerous complex heuristics used in many of the stochastic models make it impractical to implement every model of interest. Furthermore, the exponential time and space complexity of the three Markov chain models for general task graphs make them impractical for the programs we study (see Figure 1). Thus, we implemented three models for our study: the Mak and Lundstrom model, and two versions of the Kruskal and Weiss model, one using estimates of the actual variance of task execution times and another assuming task times are exponentially distributed. We refer to these models as ML, KW_{actual} and KW_{exp} respectively. Of our five programs, these models can only be used for the three fork-join programs, viz., `MP3D`, `PSIM` and `LocusRoute`.

In addition, we can infer the accuracy of the Markov Chain models from the results for ML and KW_{exp} under the following two conditions [Adve 1993]. First, ML would be equivalent to the Markov chain models (for series-parallel task graphs) if we eliminate the scheduling imprecision in ML by applying it to the condensed graph. Second, KW_{exp} would be equivalent to the Markov chain models when the two simplifying assumptions of Kruskal and Weiss are satisfied by the program, specifically, (a) the task scheduling matches the assumptions of Kruskal and Weiss, and (b) the tasks of a phase have approximately equal *mean* times and therefore can be accurately represented by i.i.d. tasks.

The inputs for the stochastic models evaluated here were derived from the inputs measured for the deterministic task graph model (described in Section 4.1). The variance of communication time for the tasks cannot be measured easily, and was instead estimated using a model described in [Adve and Vernon 1993; Adve 1993].

We begin by comparing the efficiency of the various models. Efficiency is not an issue for the Kruskal and Weiss model, which has a simple closed-form solution. In contrast, for the ML model, the model solution time and particularly memory consumption proved prohibitively high [Adve 1993] for all except the smallest task graphs. The only way we were able to apply this model was to use the condensed



(a) MP3D (b) PSIM (c) LocusRoute

Fig. 13. Comparisons between our Deterministic model and previous stochastic models

task graph. (This is only practical for statically scheduled programs, and we could apply the ML model only to `Polyroots` and `MP3D`, and not to `LocusRoute`.) Finally, the models of Thomasian and Bay, Mohan, and Kapelnikov et al. each have much higher space and time complexity than the ML model. Overall, we conclude that in most cases, these four most general stochastic models can only be applied in practice using the condensed graph as input.

The percentage errors in program execution time predicted by the models for each of the three programs, for the larger, more realistic, input size, are shown in Figure 13. The smaller input yielded qualitatively similar results in each case, but with higher errors [Adve 1993].

The results for `MP3D` and `PSIM` show that the ML model has very large errors when used with the condensed task graph. This follows because each process in this model executes only one exponentially distributed "task" of the condensed graph per phase. The high variance of the exponential distribution therefore predicts a very high synchronization delay at the barrier following each phase. These results show that using the condensed graph with exponential task models can lead to unacceptable errors.

The KW_{actual} model is consistently accurate for `MP3D`, somewhat less consistently accurate for `PSIM`, and relatively inaccurate for `LocusRoute`. The accuracy in `MP3D` and `PSIM` follows because the task time variance in the model can capture both, the variance of individual task times, as well as the small variation in mean task times across the tasks of each phase. The model also captures the static loop scheduling in `MP3D` accurately. The errors in `PSIM` and `LocusRoute` both arise because the scheduling assumptions in the model cannot capture systematic patterns of load-imbalance due to task ordering. Thus, in `PSIM`, the model does not capture the unequal amounts of work allocated to the even and odd numbered processors by processor-splitting. In `LocusRoute`, the model does not capture the load-imbalance due to an unusually large task that is executed close to the end of each phase.

The KW_{exp} model significantly overestimates the execution time for `MP3D`. The comparative accuracy of the KW_{actual} model for the same program shows that the error in KW_{exp} is due to the exponential task assumption. In both `MP3D` and `PSIM`, the errors for KW_{exp} are much smaller than the corresponding errors seen for ML because KW_{exp} uses the original task graph, which has many more tasks

than the condensed graph. This effect is even more apparent in PSIM, which has a very large number of tasks per process per phase, and KW_{exp} is almost identical to KW_{actual} for this program. Finally, for LocusRoute, KW_{exp} estimates even lower synchronization costs (and hence execution time) than KW_{actual} because, in LocusRoute, the actual variance of task times *across* the tasks in each phase is even higher than exponential. Overall, the KW_{exp} model shows that with the original task graph, the exponential task assumption leads to inconsistent accuracy, at best.

Finally, if it were practical to use the three Markov Chain models with the full task graph, they would have similar errors to KW_{exp} for MP3D (because MP3D meets the two conditions above for KW_{exp} to be equivalent to the general models). The detailed models would be more consistently accurate for PSIM and LocusRoute, because they would represent the task scheduling accurately. To use these models in practice, however, we would require more practical techniques to solve the Markov chain models with the full task graph.

6.3 Summary: When are Stochastic or Deterministic Models Preferable?

The experimental results above showed that stochastic models were successful for fork-join programs under certain scheduling assumptions (listed below). In contrast, stochastic models for non-fork-join task graphs (such as [Mak and Lundstrom 1990; Thomasian and Bay 1986; Mohan 1984; Kapelnikov et al. 1989]), which are all based on exponentially distributed task times, are too inefficient to use even for relatively small task graphs,[8] and can yield large errors with the condensed task graph. Furthermore, the polynomial-time ML model (as well as the more recent model described in [Liang and Tripathi 2000]) can only capture product-form task scheduling disciplines such as processor-sharing or FIFO, or static task scheduling using the condensed task graph.

Based on these observations, we categorize parallel programs into three groups:

(FJSimple) Programs with a fork-join synchronization structure and task scheduling disciplines where the performance is insensitive to the specific ordering of tasks, e.g., with very uniform task times as in MP3D or approximations to processor-sharing scheduling.

(FJComplex) Programs with fork-join task graph structures, but with task scheduling disciplines that do not satisfy the criterion for **FJSimple**, e.g., as in LocusRoute and PSIM.

(NonFJ) Programs with non-fork-join task graph structures such as Polyroots and DynProg.

(Programs with some very specific non-fork-join task graph structures such as divide-and-conquer or pipelined graphs can be included in the first two categories when drawing the conclusions below, but not general series-parallel programs.)

For programs in category **FJSimple**, either our deterministic model or stochastic models based on the mean and variance of i.i.d. task times [Kruskal and Weiss

[8] While sampling of possible executions could be used to reduce the solution cost, this would still be relatively expensive because each sample would require roughly the same solution time as the deterministic model.

1985; Madala and Sinclair 1991; Ammar et al. 1990] are applicable. These stochastic models may be preferable because of their simplicity and (often) closed-form solutions. For the second category, these stochastic models are again applicable, but our deterministic model is more consistently accurate because it models complex task scheduling more precisely. For all other programs (i.e., the third category), practical stochastic models that can be used for realistic program sizes and task scheduling methods do not appear to exist, whereas our model again appears consistently accurate.

7. CONCLUSIONS

In this paper, we have proposed and validated an analytical model for parallel program performance prediction, and presented several examples to illustrate that the model can be used to understand the impact of complex design changes and improve the performance of real programs. The model we propose is applicable to programs with arbitrary task graphs and a wide range of task scheduling methods. Our validation experiments with five realistic shared-memory programs showed that the model is both efficient and extremely accurate, even for programs with relatively large and complex task graphs, sophisticated task scheduling methods, highly variable task times, and significant resource contention. The errors in the execution time estimates from the model are typically less than 5%.

Several further experiments also illustrated the usefulness of the model. In experiments with two programs, insights from the model suggested design changes to improve load-balancing and accurately predicted the performance impact of the design changes. In a third program, novel detailed metrics from the model were used to explore the impact of design changes that improve communication locality as well as load-balancing. Overall, we believe these results indicate that the deterministic model can be a useful tool for evaluating sophisticated parallel program design choices analytically, using the task graph abstraction, the separate representation of task scheduling, and values of task-level parameters.

Two key features of our approach make a general, accurate and efficient model possible. First, the model is based on a powerful representation of the inherent parallelism structure in a program, the task graph. Second, the model assumes that task execution times are deterministic quantities, which permits an efficient and straightforward solution based on critical path analysis, modified to account for task scheduling precisely and to evaluate the average costs of communication and resource contention at every step of the analysis. The model accurately represents key details of task scheduling, the order of task execution, non-uniform task times, and average communication costs. A quantitative comparison with representative stochastic models showed the relative benefits of those models and our deterministic model, and led us to categorize programs into three classes. For the simplest of the three classes, namely, fork-join programs with restricted task scheduling disciplines, simple stochastic models such as [Kruskal and Weiss 1985] were practical, efficient, and required less detailed input information than the deterministic model. For the other two classes of programs, which include four of the five programs in this study, existing stochastic models appeared inaccurate or impractical.

There are some potentially significant limitations to the deterministic task graph analysis approach. First, it provides detailed numerical predictions rather than

simple, intuitive insights provided by simpler closed form models. In our experience from this and other work [Adve et al. 2000], however, the process of constructing a task graph and other input information for an application does lead to valuable insights about the details of program structure and performance. Second, the model requires a fairly detailed input program description whereas simpler information suffices for some programs with simple task graphs and scheduling behavior where simpler models may apply.

The third limitation of our model, shared by previous program models, is that it does not provide the means *to predict how model parameters vary* for different systems, task scheduling methods, or other program changes. Several techniques (mostly compiler-driven) exist for deriving intrinsic parameters that can be used to predict how computational costs [Sarkar 1989; Balasundaram et al. 1991; Fahringer and Zima 1993; Wang 1994] and parallelism [Kumar 1988; Larus 1993; Parashar et al. 1994] vary as a function of parameters such as problem size or number of processors. It is much more challenging to predict how shared memory communication parameters (e.g., cache miss rates) vary as a function of such changes or changes to the task scheduling algorithm, and currently this must be done using a manual, algorithmic analysis of each parallel program [Culler et al. 1993; Tsai and Agarwal 1993; Harzallah and Sevcik 1995]. To fully exploit the predictive capabilities of our model, we would like to be able to predict how the communication parameters vary by using some *intrinsic description of communication behavior*, i.e., a description that is independent of system size or task scheduling (just as the task graph provides an intrinsic description of parallelism structure). This remains a difficult but important challenge for future research.

The experiments and results in this paper also suggest some other possibilities for future research. The model would be much more useful in practice if the process of deriving model inputs can be partially or fully automated. Significant research issues will arise in developing the compiler and operating system infrastructure required to derive the task graph and measure the requisite model parameters. (For example, in recent work, we have developed compiler techniques to extract task graphs automatically for compiler-parallelized message passing programs [Adve and Sakellariou 2000], but extending these techniques to broader classes of programs requires further research.) If successful, however, such an integrated package combining the complementary strengths of analytical studies and measurement or simulation should yield a comprehensive and powerful tool for parallel program performance evaluation and prediction.

REFERENCES

Adve, V. and Sakellariou, R. 2000. Compiler Synthesis of Task Graphs for a Parallel System Performance Modeling Environment. In *Proc. 13th Int'l Workshop on Languages and Compilers for High Performance Computing (LCPC '00)*, Yorktown Heights, NY.

Adve, V. S. 1993. Analyzing the Behavior and Performance of Parallel Programs. Ph.D. thesis, University of Wisconsin-Madison.

Adve, V. S., Bagrodia, R., Browne, J. C., Deelman, E., Dube, A., Houstis, E., Rice, J. R., Sakellariou, R., Sundaram-Stukel, D., Teller, P. J., and Vernon, M. K. 2000. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *IEEE Trans. on Software Engineering (Special Issue on Software and Performance) 26,* 11 (Nov.), 1027–1048.

ADVE, V. S. AND VERNON, M. K. 1993. The Influence of Random Delays on Parallel Execution Times. In *Proc. 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 61–73.

ADVE, V. S. AND VERNON, M. K. 1994. Performance Analysis of Mesh Interconnection Networks with Deterministic Routing. *IEEE Transactions on Parallel and Distributed Systems 5,* 3 (March), 225–246.

ALEXANDROV, A., IONESCU, M., SCHAUSER, K. E., AND SCHEIMAN, C. 1995. LogGP: Incorporating Long Messages into the LogP Model. In *Proc. 7th Annual Symp. Parallel Algorithms and Architecture*.

AMDAHL, G. M. 1967. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings*, Volume 30, pp. 483–485.

AMMAR, H. H., ISLAM, S. M. R., AMMAR, M., AND DENG, S. 1990. Performance Modeling of Parallel Algorithms. In *Proc. 1990 Int'l Conf. on Parallel Processing*, pp. III 68–71.

BALASUNDARAM, V., FOX, G., KENNEDY, K., AND KREMER, U. 1991. A Static Performance Estimator to Guide Data Partitioning Decisions. In *Proceedings of the Third ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Williamsburg, VA.

BLUMOFE, R. D., JOERG, C. F., LEISERSON, C. E., RANDALL, K. H., , AND ZHOU, Y. 1995. Cilk: An efficient multithreaded runtime system. In *Proc. 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Santa Barbara, CA, pp. 207–216.

BREWER, E. A. 1995. High-Level Optimization via Automated Statistical Modeling. In *Proc. 5th ACM Symp. Principles and Practice of Parallel Programming*.

BROOKS III, E. D. 1988. PCP: A Parallel Extension of C that is 99% Fat Free. Tech. rep. (September), Lawrence Livermore National Laboratory.

BROWNE, J. C., HYDER, S. I., DONGARRA, J., MOORE, K., AND NEWTON, P. 1995. Visual Programming and Debugging for Parallel Computing. *IEEE Parallel and Distributed Technology 3,* 1 (Spring).

BROWNE, S., DONGARRA, J., GARNER, N., LONDON, K., AND MUCCI, P. 2000. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proc SC'2000*.

CROVELLA, M. E., LEBLANC, T. J., AND MEIRA, W. 1995. Parallel Performance Prediction Using the Lost Cycles Toolkit. Tech. Rep. TR580, Dept. Computer Science, U. Rochester.

CULLER, D., KARP, R., PATTERSON, D., SAHAY, A., SCHAUSER, K. E., SANTOS, E., SUBRAMONIAN, R., AND VON EICKEN, T. 1993. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. 4th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*.

CVETANOVIC, Z. 1987. The Effects of Problem Partitioning, Allocation and Granularity on the Performance of Multiple-Processor Systems. *IEEE Trans. on Computers C-36,* 4, 421–432.

DIKAIAKOS, M. 1994. Functional Algorithm Simulation. Ph.D. thesis, Department of Computer Science, Princeton University.

DIKAIAKOS, M., ROGERS, A., AND STEIGLITZ, K. 1994. FAST: A Functional Algorithm Simulation Testbed. In *International Workshop on Modelling, Analysis and Simulation of Computer and Telecommunication Systems – Macsots '94*, pp. 142–146.

DUBOIS, M. AND BRIGGS, F. A. 1982. Performance of Synchronized Iterative Processes in Multiprocessor Systems. *IEEE Trans. on Software Engineering SE-8,* 4 (July), 419–431.

EAGER, D. L., ZAHORJAN, J., AND LAZOWSKA, E. D. 1989. Speedup versus Efficiency in Parallel Systems. *IEEE Trans. on Computers C-38,* 3 (March), 408–423.

FAHRINGER, T. 1993. Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers. Ph.D. thesis, U. Vienna.

FAHRINGER, T. AND ZIMA, H. 1993. A Static Parameter-Based Performance Prediction Tool for Parallel Programs. In *Proc. 1993 ACM Int'l. Conf. on Supercomputing*, Tokyo.

FLATT, H. 1984. A Simple Model of Parallel Processing. *IEEE Computer 17*, 95.

FLATT, H. AND KENNEDY, K. 1989. Performance of Parallel Processors. *Parallel Computing 12*, 1–20.

Frank, M., Vernon, M., and Agarwal, A. 1997. LoPC: Modeling Contention in Parallel Algorithms. In *Proc. 6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Las Vegas, USA.

Gross, D. and Harris, C. M. 1985. *Fundamentals of Queueing Theory* (2nd ed.). John Wiley and Sons.

Gustafson, J. L. 1988. Reevaluating Amdahl's Law. *Communications of the ACM 31,* 5 (May).

Hack, J. 1989. On the Promise of General-purpose Parallel Computing. *Parallel Computing 10*, 261–275.

Hartleb, F. and Mertsiotakis, V. 1992. Bounds for the Mean Runtime of Parallel Programs. In *Proceedings of the Sixth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pp. 197–210.

Harzallah, K. and Sevcik, K. C. 1995. Predicting Application Behavior in Large-Scale Shared Memory Multiprocessors. In *Proc. Supercomputing '95*, San Diego, CA.

Heidelberger, P. and Trivedi, K. S. 1983. Analytic Queueing Models for Programs with Internal Concurrency. *IEEE Trans. on Computers C-32,* 1 (January), 73–82.

Horowitz, E. and Sahni, S. 1984. *Fundamentals of Computer Algorithms.* Computer Science Press International, Inc., Rockville, MD.

Jonkers, H., van Gemund, A. J., and Reijns, G. L. 1995. A Probabilistic Approach to Parallel System Performance Modelling. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, pp. II (Software Technology).

Kapelnikov, A., Muntz, R. R., and Ercegovac, M. D. 1989. A Modeling Methodology for the Analysis of Concurrent Systems and Computations. *Journal of Parallel and Distributed Computing 6*, 568–597.

Kruskal, C. P. and Weiss, A. 1985. Allocating Independent Subtasks on Parallel Processors. *IEEE Trans. on Software Engineering SE-11,* 10 (October), 1001–1016.

Kumar, M. 1988. Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications. *IEEE Trans. on Computers 37,* 9 (Sept.), 1088–1098.

Larus, J. R. 1993. Loop-Level Parallelism in Numeric and Symbolic Programs. *IEEE Trans. on Parallel and Distributed Systems 4,* 7 (July), 812–826.

Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. 1984. *Quantitative System Performance.* Prentice-Hall.

Lewandowski, G., Condon, A., and Bach, E. 1996. Asynchronous Analysis of Parallel Dynamic Programming Algorithms. *IEEE Trans. Parallel and Distributed Systems 7,* 4 (April), 425–438.

Liang, D.-R. and Tripathi, S. K. 2000. On Performance Prediction of Parallel Computations with Precedence Constraints. *IEEE Trans. on Parallel and Distributed Systems 11,* 5 (May), 491–508.

Madala, S. and Sinclair, J. B. 1991. Performance of Synchronous Parallel Algorithms with Regular Structures. *IEEE Trans. Parallel and Distributed Systems 2,* 1 (Jan.), 105–116.

Mak, V. W. and Lundstrom, S. F. 1990. Predicting Performance of Parallel Computations. *IEEE Trans. on Parallel and Distributed Systems 1,* 3 (July), 257–270.

Mendes, C. and Reed, D. 1998. Integrated Compilation and Scalability Analysis for Parallel Systems. In *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques.*

Mohan, J. 1984. Performance of Parallel Programs: Model and Analyses. Ph.D. thesis, Carnegie Mellon University.

Narendran, B. and Tiwari, P. 1992. Polynomial Root-Finding: Analysis and Computational Investigation. In *Proc. 4th Annual Symp. on Parallel Algorithms and Architectures.*

Parashar, M., Hariri, S., Haupt, T., and Fox, G. 1994. Interpreting the Performance of HPF/Fortran 90D. In *Proceedings of Supercomputing '94*, Washington, D.C.

Sarkar, V. 1989. Determining Average Program Execution Times and their Variance. In *Proc. 1989 SIGPLAN Conference on Programming Language Design and Implementation.*

Schopf, J. M. 1997. Structural prediction models for high-performance distributed applications. In *Proc. Cluster Computing Conference*, Atlanta, USA.

Schopf, J. M. and Berman, F. 2001. Using stochastic information to predict application behavior on contended resources. *Int'l Journal of Foundations of Computer Science 12,* 3 (Jun), 341–364.

Singh, J. P., Weber, W.-D., and Gupta, A. 1992. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News 20,* 1 (March), 5–44.

Sundaram-Stukel, D. and Vernon, M. 1999. Predictive Analysis of a Wavefront Application Using LogGP. In *Proc. 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Atlanta, USA.

Thomasian, A. and Bay, P. F. 1986. Analytic Queueing Network Models for Parallel Processing of Task Systems. *IEEE Trans. on Computers C-35,* 12 (December), 1045–1054.

Towsley, D., Rommel, G., and Stankovic, J. A. 1990. Analysis of Fork-Join Program Response Times on Multiprocessors. *IEEE Trans. Parallel and Distributed Systems 1,* 3.

Tsai, J. and Agarwal, A. 1993. Analyzing Multiprocessor Cache Behavior Through Data Reference Modeling. In *Proc. 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*.

Tsuei, T.-F. and Vernon, M. K. 1990. Diagnosing Parallel Program Speedup Limitations Using Resource Contention Models. In *Proc. 1990 Int'l Conf. Parallel Processing*.

Tsuei, T.-F. and Vernon, M. K. 1992. A Multiprocessor Bus Design Model Validated by System Measurement. *IEEE Trans. on Parallel and Distributed Systems 3,* 6, 712–727.

van Gemund, A. J. 1996. Performance Modeling of Parallel Systems. Ph.D. thesis, Delft University of Technology.

van Gemund, A. J. 2003. Symbolic Performance Modeling of Parallel Systems. *IEEE Trans. on Parallel and Distributed Systems 14,* 2 (Feb.), 154–165.

Vernon, M. K., Lazowska, E. D., and Zahorjan, J. 1988. An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols. In *Proc. 15th International Symp. on Computer Architecture*.

Vrsalovic, D. F., Siewiorek, D. P., Segall, Z. Z., and Gehringer, E. F. 1988. Performance Prediction and Calibration for a Class of Multiprocessors. *IEEE Trans. on Computers 37,* 11 (November), 1353–1365.

Wang, K.-Y. 1994. Precise Compile-Time Performance Prediction for Superscalar-Based Computers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, pp. 73–84.

Willick, D. L. and Eager, D. L. 1990. An Analytic Model of Multistage Interconnection Networks. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 192–202.

Xu, Z., Zhang, X., and Sun, L. 1996. Semi-Empirical Multiprocessor Performance Predictions. *Journal of Parallel and Distributed Computing 39,* 1 (Jan.).

Yazici-Pekergin, N. and Vincent, J.-M. 1991. Stochastic Bounds on Execution Times of Parallel Programs. *IEEE Trans. on Software Engineering 17,* 10 (October), 1005–1012.