A Deterministic Model for Parallel Program Performance Evaluation

Vikram S. Adve
Rice University
and
Mary K. Vernon
University of Wisconsin-Madison

Analytical models for parallel programs have been successful at providing simple qualitative insights and bounds on scalability, but have been less successful in practice for predicting detailed, quantitative information about program performance. We develop a conceptually simple model that provides detailed performance prediction for parallel programs with arbitrary task graphs, a wide variety of task scheduling policies, shared-memory communication, and significant resource contention. Unlike many previous models, our model assumes deterministic task execution times which permits detailed analysis of synchronization, task scheduling, the order of task execution, as well as mean values of communication costs. The assumption of deterministic task times is supported by a recent study of the influence of non-deterministic delays in parallel programs. We show that the deterministic task graph model is accurate and efficient for five 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. We also 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, further novel metrics are used to obtain insight into the impact of program design changes that improve communication locality as well as load-balancing. Finally, we briefly present results of a comparison between our model and representative models based on stochastic task execution times.

Categories and Subject Descriptors: []:

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

1. INTRODUCTION

Researchers have successfully developed simple analytical models for obtaining qualitative insights and bounds on the scalability of parallel programs as a function of input and system size [Amdahl 1967; Gustafson 1988; Culler et al. 1993; Frank et al. 1997]. In contrast, tools for more detailed analysis of program performance have been primarily based on measurement and simulation rather than on analytical

This work is supported in part by the National Science Foundation (CDA-9024618, CDA-8920777, CCR-9024144), by DARPA Contract No. N66001-97-C-8533, and by an IBM Graduate Fellowship.
Author's addresses: Vikram S. Adve, Computer Science Department, MS 132, Rice University, 6100 Main St., Houston TX 77005-1892. Mary K. Vernon, University of Wisconsin-Madison, Computer Sciences Department, 1210 West Dayton St., Madison WI 53706.

modeling.

This paper develops an analytical model that can complement measurement and simulation techniques, and is of significant practical use for three reasons. First, it provides a level of abstraction that is useful for gaining insight into key program performance issues. Second, it supports an efficient technique for computing program execution times. Third, the analytical model provides the capability to accurately predict the performance of a given program on future systems or system configurations, and also accurately predicts the performance impact of various program design changes *before* implementing the changes in the code.

Previous analytical models that were designed for detailed performance analysis of parallel programs are reviewed briefly in Section 6. To our knowledge, only a few of the previous models have been evaluated using realistic applications. One of the contributions of this research has been to evaluate representative examples of these models on a set of realistic applications to understand the previous state of the art.

Many previous analytical models are *stochastic* models, in which the execution time of the tasks of the program are represented as non-deterministic quantities. Our results show that stochastic models can be simple, accurate, and highly efficient for programs with simple fork-join synchronization behavior (i.e., alternating serial and parallel phases), restricted types of task scheduling, and that don't have bottlenecks that are due to the specific order of task execution [Dubois and Briggs 1982; Kruskal and Weiss 1985; Ammar et al. 1990]. However, stochastic models that apply to programs with more complex behavior use complex and heuristic solution techniques and assume that the tasks in a program have exponentially distributed execution times to permit tractable solutions [Mohan 1984; Thomasian and Bay 1986; Mak and Lundstrom 1990; Kapelnikov et al. 1989]. As explained further in section 6, these models are expensive to solve, and the assumption of exponential task execution times leads to poor or inconsistent accuracy in the model predictions. We believe these limitations are inherent in stochastic models because of the complexity of predicting synchronization costs and task scheduling behavior in such models, except for fork-join programs.

The inherent limitations in stochastic models provide strong motivation for considering the alternative, namely analytical models that assume deterministic task execution times. A common perception is that deterministic models cannot predict synchronization costs accurately, since there are several sources of nondeterminism in parallel task execution times. For the purposes of predicting the execution time of a particular application of interest, however, there is evidence to the contrary. Specifically, in a recent study, we provided strong experimental evidence as well as formal and intuitive justification that for many (shared memory) parallel applications it is reasonable to *ignore the variance of task execution times due to variable communication costs* when predicting the application execution time [Adve and Vernon 1993]. These results, discussed in more detail in Section 2.2, suggest that a deterministic model might be a viable alternative to stochastic models for parallel program performance prediction.

A few previous analytical models assume deterministic task execution times [Vrsalovic et al. 1988; Cvetanovic 1987; Tsuei and Vernon 1990; Harzallah and Sevcik 1995]. Although the authors do not cite any direct justification for the deterministic assumption and these previous models only apply to programs with simple fork-join synchronization (as explained in Section 6), the models have each been shown to be both accurate and efficient for several parallel programs to which they apply.

In this paper, we propose a simple, widely applicable, *deterministic* model for parallel program performance prediction, based on an approach we call deterministic task graph analysis. The inputs to the model are: 1) a task-graph that describes the synchronization behavior of an application, 2) a description of the task scheduling algorithm used to allocate tasks to processors, and 3) parameters describing the processing time and average communication rates of each task. In contrast to previous models, we use a solution technique based on modified critical path analysis, which applies to parallel programs with arbitrary task graphs and a wide range of static and dynamic task scheduling methods. This approach provides detailed insights into the impact of task synchronization and task scheduling on program performance. The model also predicts the 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 (shared memory) parallel applications on realistic input data. For the applications we have examined, we find that the model is efficient even for programs with large and complex task graphs. More importantly, it is *consistently accurate* 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.

Finally, we use three 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 metrics to obtain insight into and explore the impact of sophisticated program design changes that improve communication locality as well as load-balancing.

The remainder of the paper is organized as follows. In Section 2.2, we define some key terms that are used throughout the paper, and explain the motivation for a deterministic model in more detail. In Section 3, we describe the deterministic task graph analysis method for parallel program performance prediction. In Section 4, we evaluate the accuracy, efficiency and applicability of deterministic task graph analysis using five parallel programs. In Section 5, we illustrate the use of our model for evaluating program design tradeoffs. In Section 6, we review previous analytical models and briefly compare the efficiency and accuracy of our model with those of representative stochastic models. Section 7 presents the conclusions from this work and 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. To illustrate some of these terms, the task graphs for five shared-memory programs are shown in Figure 1. The applications, inputs, task graphs, and scheduling functions for these programs

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. Sometimes called a *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].

are described in Section 4.

The task graph provides an abstract but precise representation of the parallelism and synchronization in a program. Perhaps the most important goal of our definitions is that, ideally and whenever possible in practice, the task graph should be a representation of the inherent parallelism in the program, *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. A typical example is that the tasks are the iterations of a loop, and the scheduling function specifies how the iterations are scheduled onto the processes that execute the program. In some shared memory programs, process initialization 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 such as those described for the application `LocusRoute` in Section 5.3.

The condensed task graph is a compact version of the task graph that can be used when the allocation of tasks to processes can be precomputed (such as for static task scheduling algorithms). The tradeoffs in using the condensed task graph are described in [Adve 1993]. This definition will also be useful to understand some of the properties of previous analytic models. 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 are based on graph models that can be viewed as equivalent to either the task graph or the condensed task graph.

Throughout the remainder of the paper we use N to denote the number of tasks in a program or program phase, $P_{\max}$ to denote the maximum parallelism (the maximum number of active processors in any execution of the program or program phase), and P to denote the number of processors under consideration.

2.1 A Framework for Parallel Program Performance Prediction Models

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 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 such as communication costs, interconnection network 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 one model uses additional parameters computed from the higher-level model to account for this effect approximately [Mak and Lundstrom 1990].

2.2 Motivation for a Deterministic Model

A model with stochastic task execution times essentially represents a program as a stochastic process in which each state is some combination of tasks in execution, even if this representation is not explicitly constructed in the model solution. Average synchronization costs in such a model represent the average across all possible

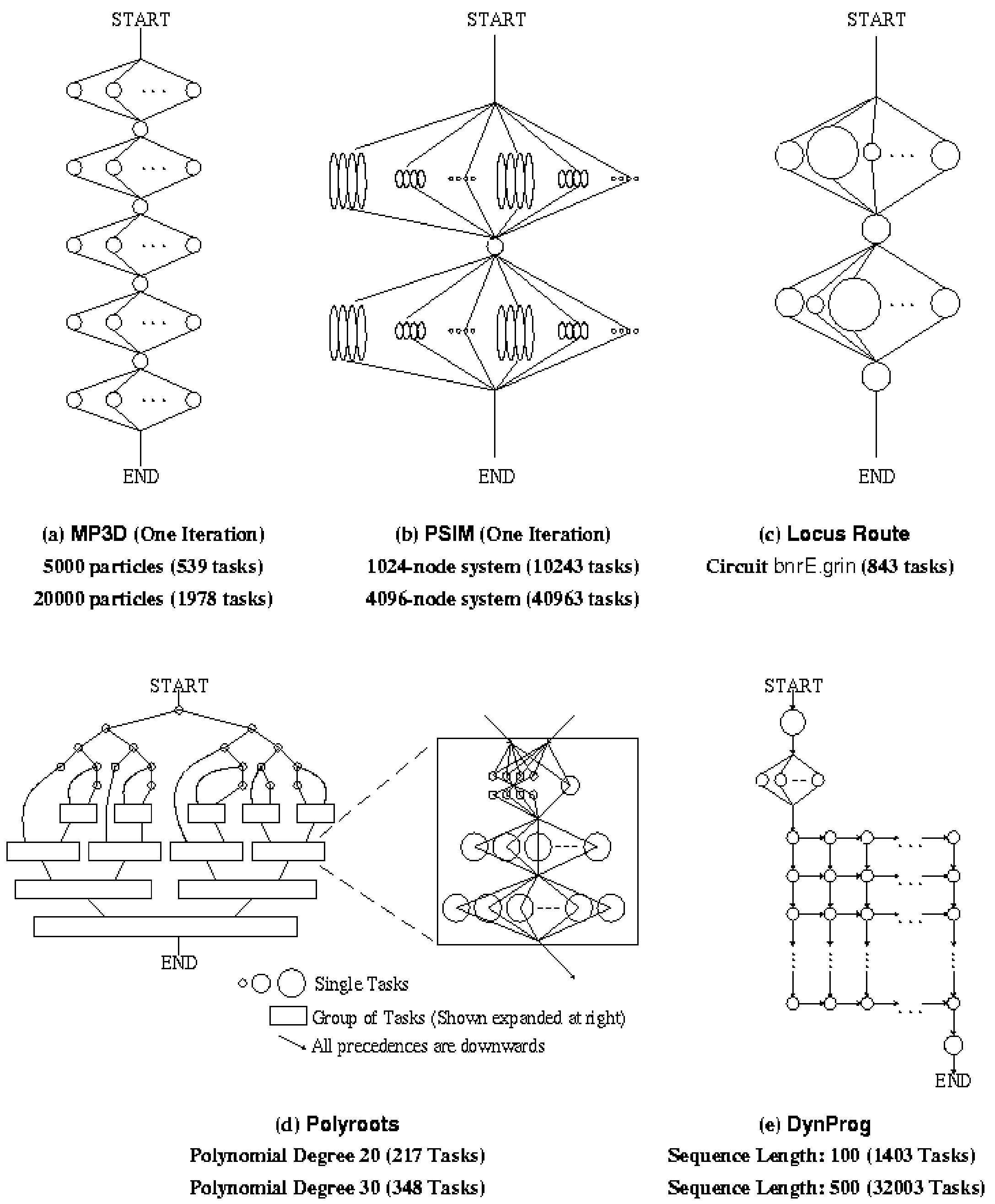


Fig. 1. Task Graphs for Five Shared Memory Applications.
START and END denote dummy tasks marking the entry and exit for each graph.

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 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. In a recent study of the accuracy and efficiency of some representative examples of such models, we showed that models in the latter class are very 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 are briefly reviewed 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.

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. However, current stochastic models based on such statistics are restricted to fork-join programs with limited types of task scheduling, as described in in Section 6.

A second fundamental limitation of a deterministic model is that it cannot account for any possible variance in *individual* task execution times, which could cause the model to underestimate synchronization costs. We can identify four possible sources of variability in individual task execution times. First, variability can arise due to multiprogramming of the processors, but the impact of this variability must be considered in the analysis rather than in the representation of task execution time. Such analysis is outside the scope of our work, i.e., we focus on a program executing stand-alone or on a dedicated partition of a multiprocessor system. Second, program behavior is generally different on different inputs, and this must be accounted for in models that predict behavior across different program inputs (e.g., scalability models such as Vrsalovic et al. [Vrsalovic et al. 1988]). Like most other models mentioned here, however, we focus on predicting the execution time of a program for a single input, and this source of variability is not relevant. Note, however, that these models could still be used to predict program scalability by separately analyzing how model input parameters scale with problem size.

The two other sources of variability are potentially more important within the context of evaluating a "stand-alone" program on a single input. These are 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].

First, 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 granularity similar to those on current systems, 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. (These results hold for large-scale shared-memory systems as well as single-bus systems like the Sequent Symmetry.) 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 (across different executions), 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, one of the programs we study exhibits this behavior, and the accuracy and usefulness of our model for this program is studied in Sections 4 and 5 respectively. In general, however, we believe such behavior to be fairly uncommon in parallel applications.

Based on these arguments and experiments, we hypothesize that the potential simplicity of a deterministic model which ignores variance in task execution times could make any loss of accuracy 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 contention 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. 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

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

are fixed, independent of the other tasks that may be simultaneously in execution (e.g., communication latency if contention can be ignored). 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 to compute task delays that vary with the number of processors and/or the other tasks that are executing concurrently, e.g, total communication costs including contention. The number and type of resource usage parameters depend on the choice of system-level model used, and can vary from one system to another. Typical examples of resource usage parameters are the task's cache miss rate and the fraction of cache misses that access dirty cache lines on a invalidation-based cache coherent shared memory system.

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.[1] 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, however, is very general and not intended as an input representation in practice. Instead, many scheduling functions can be described in a common scheduling framework in our implementation of the model. This issue is discussed further in Section 3.4.

Henceforth, we assume that only one process per processor is used during the execution of the program, as in many parallel programs today. In [Adve 1993], we discuss an extension to model multiple processes per processor.

3.2 The High-Level Model of Task Execution

The high-level model component of our model essentially performs three key functions:

(1) a simple graph traversal algorithm to enforce task precedences;

(2) evaluating the scheduling function to model the impact of task allocation and task execution ordering when there are fewer processors than the maximum 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.

[1] Note that internal data within each task may also be considered when performing task scheduling even though such parameters are not made explicit in the definition.

Figure 3.2 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 resource contention. The basic algorithm can be determined from the figure by ignoring the first highlighted step which computes the values D_i, and by assuming $D_i \equiv 0, 1 \le i \le N$ 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, regardless of the number of processors used to execute the application. 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, 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 $\mathcal{E}$ 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, is the essence of deterministic task graph analysis. Unlike previous detailed analytical models in the literature (discussed in Section 6), this approach is conceptually simple. For example, 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 an important advantage of the deterministic approach.

The Complete Model Including Overhead Costs

For a model to apply to most programs, it must account for resource contention and other overheads that depend on the number of processors and/or the set of tasks that are executing concurrently. In our 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

Fig. 2. The Complete Deterministic Model Including Variable Overhead Costs
(The basic model can be derived by setting $D_i \equiv 0$ in the highlighted steps.)

Inputs All inputs listed in Table 3.1.

Algorithm

```
r_i ← T_i, 1 ≤ i ≤ N                      /* Remaining cpu requirement for task i */
ctr_i ← #Parents of i, 1 ≤ i ≤ N
ℰ ← { 1 }                                 /* Initial set of executing tasks */
L ← { }                                   /* Initial set of ready tasks waiting to be scheduled */
T_total ← 0                               /* Elapsed time since start of program */

Do until ℰ empty {                        /* At most N times; exactly N if
                                             no two tasks complete simultaneously */
   1) [Compute D_i(ℰ, {M_i,j}, r) ∀ i ∈ ℰ using the lower-level system model]

   2) [T_elapse ← min{r_i + D_i : i ∈ ℰ}]       /* Time till next task completion */
      [𝒞 ← {j ∈ ℰ : r_j + D_j = T_elapse }]    /* Set of tasks that complete next */
      ℰ ← ℰ − 𝒞

   3) T_total ← T_total + T_elapse
      [r_i ← r_i − T_elapse × r_i/(r_i+D_i), ∀i ∈ ℰ]

   4) For each task x ∈ 𝒞
         For each immediate successor c of task x
            ctr_c ← ctr_c − 1
            If ctr_c = 0                   /* I.e., if all parents of task c have completed */
               L ← L ⋃ {c}                 /* Add task c to list of ready tasks */

   5) For each idle process p
         j ← Sched(L, p)
         if j > 0
            L ← L − {j}
            ℰ ← ℰ ⋃ {j}
}

Total Program Execution Time = T_total
```

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 3.2.

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 such sets in deterministic task graph analysis is at most N. 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, given by $\frac{r_i}{r_i+D_i}$. Then, for each such task i, $T_{elapse} \times \frac{r_i}{r_i+D_i}$ 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. Our experiments with statically scheduled programs have corroborated this claim. This property is important 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 actual computational cost of applying the basic and full model to several realistic programs is studied in Section 4. Here, we focus on the computational complexity of the solution algorithms.

For a graph with N nodes, E edges, and maximum parallelism $P_{\max}$, the overall complexity of the basic algorithm is $O(NP_{\max} + E + N)$. 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 the third step, each edge in the graph needs to be examined exactly once in the overall solution. The cost of the fourth step depends on the cost of evaluating the scheduling function. In any case, at most $NP_{\max}$ evaluations of the function need to be made, 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 algorithm 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(P) for a queueing network with P 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]), and assuming that the number of queueing centers and the number of iterations per MVA solution are small.

Nevertheless, in practice, the system-level solution step can dominate the overall solution time of the model. Thus, reducing the number of times the system-level model must actually be solved is invariably worthwhile. In [Adve 1993], we discuss techniques to minimize model solution time in practice.

3.3 An Example System-Level Model for the Sequent Symmetry

The actual choice of system-level model that should be used for each study depends on the system under consideration, and perhaps also on the required accuracy of the modeling study. Therefore, unlike many previous authors [Thomasian and Bay 1986; Kapelnikov et al. 1989; Mak and Lundstrom 1990; Harzallah and Sevcik 1995], 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 simple but system-specific queueing model for the bus and memory modules to calculate communication overhead costs. This is based on a model originally developed and validated for the same system by Tsuei and Vernon [Tsuei and Vernon 1992], but uses a significantly simpler approach to model the protocol limit on the number of outstanding bus requests. To validate our simplified model, we compared the model predictions to direct hardware measurements of communication costs for these applications, and found that the model predicted mean response times within 10% of the measured values in most cases, and the error was less than 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 $invalidate(inv)$, and 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

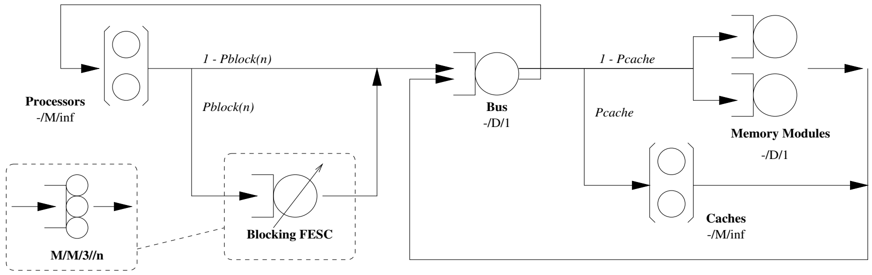


Fig. 3. Queueing network model for Sequent Symmetry multiprocessor system

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] Using a single-chain model requires common (average) values of λ_{bus}, f_{inv}, f_{rwb} and p_{cache} for all the customers. We compute λ_{bus} as $\lambda_{bus} = 1/(P_{active} \times \Sigma_{i \in \mathcal{E}} 1/\lambda_{bus}(i))$, and the other three as weighted averages, weighted by $1/\lambda_{bus}(i)$. For example, $f_{inv} = (\Sigma_{i \in \mathcal{E}} f_{inv}(i)/\lambda_{bus}(i))/\lambda_{bus}$. We use this simplification to avoid having P_{active} classes of customers, in order to keep the solution complexity to $O(|\mathcal{E}|)$ as described below.

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 certain 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.

The simplification in our model is for 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

[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].

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 first estimated by solving a separate M/M/3//n queue for each $4 \leq n \leq P_{active}$. 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, $\forall n \geq 4$. These residence times are intermediate results of the queueing network solution described below.

We used customized Mean Value Analysis to solve the queueing network, which gives fairly accurate mean response times and other metrics 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 $\forall n \geq 4$ (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 complexity of the queueing network solution is $O(|\mathcal{E}|)$, as required for the $O(NP_{\max} + E)$ bound 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 Model Inputs in Practice

The issues that arise in deriving the model inputs for a real program in practice are described in some detail in [Adve 1993]. We only briefly discuss them here. Constructing a task graph for a program is generally straightforward with a basic understanding of the parallelism and synchronization structure of the program. For the five programs whose task graphs are illustrated in Figure 1, it was not difficult to develop scripts that generated the task graph for each program for a given program input. For example, the program with the most complex synchronization pattern was *POLY* (Figure 1(d)). A large number of flags with spin waiting are used to enforce precedences in the computation at a fine granularity. Nevertheless, the precedences are uniquely determined by the program input. A careful understanding of the code was sufficient to construct an algorithm that enumerates the tasks and precedences for a specified input.

The second key input to the model is the scheduling algorithm, and here the principal question is how the algorithm can be specified to the model. A uniform scheduling framework that includes a broad class of common task scheduling functions is provided in our implementation of the deterministic task graph model. A scheduling function in this framework can be described by four components: (a) the number of task queues, (b) the initial queue to which each process is assigned, (c) the policy for initial allocation of tasks to queues (round-robin with a chunk size, or user-enumerated), and (d) the algorithm by which a process switches queues when it's previously assigned queue becomes empty. We implemented four choices of the algorithm in (d): no switching, next non-empty, next largest queue, or next queue with fewest assigned processes. In specifying the scheduling function for a program, tasks are grouped into task-groups (e.g,. the tasks of a parallel loop), and the initial allocation policy for tasks to queues (e.g., round-robin) is specified separately for each task group. This framework is enough to capture a fairly large class of static, semi-static and dynamic scheduling policies used in shared-memory codes today. Although the framework can be expanded to include other types of policies, it is inherently difficult to model policies that reschedule tasks based on intermediate results of the program.

The other inputs required for the model are the task CPU demands, and (for the full version of the model) the shared resource usage parameters. For the validations in this paper, we directly measured the task CPU demands using software timers, and the communication parameters using a hardware bus monitor. In current practice, performance studies most commonly focus on evaluating an existing program on an existing system, where such direct measurements are possible.

It is also possible to use the deterministic task graph model to predict the effect of some program or system changes on program performance, as we show in Section 6. In the general case, separate techniques would be required to predict the impact of such changes on the input communication parameters. In particular, it is not within the scope of our model to predict how these input parameters vary with system size or different task scheduling disciplines. (This restriction is shared by previous models for parallel program performance prediction.) Separate analysis techniques to predict shared-memory communication and cache miss rate parameters are available and could be used to provide inputs to our model, but these analyses are very algorithm-specific [Culler et al. 1993; Tsai and Agarwal 1993; Harzallah and Sevcik 1995]. For many types of program changes, however, useful insights about the performance impact can be obtained while ignoring the effect of the changes on the resouce usage parameters (i.e., using existing measured values). For example, in Section 6, we present examples where we used our model to predict the impact of changes in the task scheduling policy on load-balancing as well as locality.

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. 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.

4.1 Methodology Used in the Study

Our experiments were conducted on a 20-processor Sequent Symmetry S-81. Scripts to generate each of the task graphs were constructed as discussed in Section 3.1. The task CPU requirements were measured in software 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

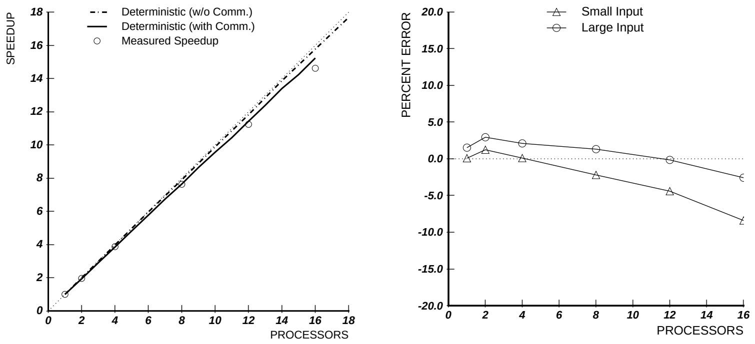
Table III. Applications Evaluated using the Analytical Models

Name	Description	Task Graph	Task Scheduling	Perf. Losses
`MP3D`	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`	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`	Wire routing in VLSI standard cells (Commercial quality)	Fork-join: two parallel phases; widely varying task sizes per phase	Dynamic allocation in each phase with single FIFO task queue	Load imbalance due to task skew; cache misses
`Poly-roots`	Compute roots of a polynomial with arbitrary precision (integer) coeffs.	Non-series-parallel; widely varying task sizes	Dynamic allocation with single FIFO task queue	Load imbalances; limited overall parallelism
`DynProg`	Dynamic programming algorithm for alignment of 2 gene sequences	Pipelined: dynamic programming array of numerous small but uniform tasks;	Static round-robin allocation of rows of tasks to processes	Limited parallelism at beginning and end

three applications that have significant communication overhead (`MP3D`, `PSIM` and `LocusRoute`), the mean communication costs on one processor were subtracted from the measured CPU requirements since communication behavior including contention would be represented separately and precisely in the full deterministic model. The specific scheduling functions used in each program are explained along with the results for the individual programs.

In measuring the resource usage parameters, 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 `MP3D` and more strongly true for `PSIM`.[3] Alternatively, either separate measurements or a separate analytical model would have to be used to derive the appro-

[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.



(a) Speedups for large input (b) Error in predicted execution time

Fig. 4. Predicted and measured performance of `MP3D`

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

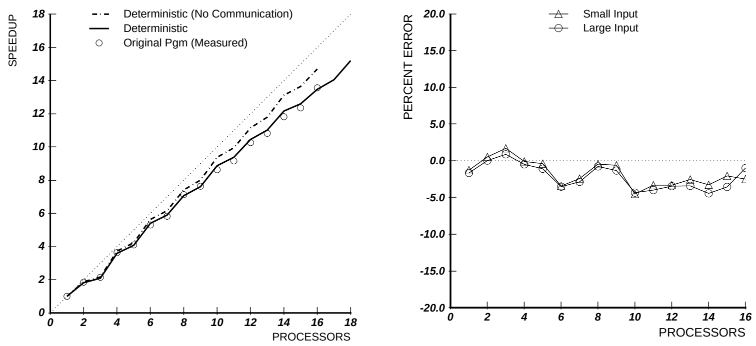
4.2 Solution Efficiency of the Model

For the five programs studied in this paper, the task graphs (with the larger inputs) ranged in size from 348 to 40963 tasks. Of these, we used the basic deterministic model for two of the programs, NW and $POLY$. For many of the programs, the model solution is virtually instantaneous. Overall, the two most expensive programs to analyze were `PSIM` and `DynProg`, with 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.

4.3 Accuracy of the Model

Results for `MP3D`

The first program, `MP3D`, is taken from the SPLASH suite of parallel applications [Singh et al. 1992]. 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



(a) Speedups for large input (b) Error in predicted execution time

Fig. 5. Predicted and measured performance of `PSIM`

hence we can consider a chunk to be a single task (see Table I). 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 (one of the scheduling methods supported directly in our model implementation).

We consider two input sizes for `MP3D`: a small input size of 5000 particles, and a somewhat more realistic input size of 20000 particles. The percentage errors in the predicted execution time from our model for different values of P are given for the two input sizes in Figure 4 (b). Figure 4 (a) shows the measured and predicted speedups for this application, including the predicted speedup ignoring communication overhead (i.e., using the basic model alone). The predicted speedups are relative to the predicted execution time on $P = 1$.

The results show that the deterministic task graph model is highly accurate for `MP3D`. 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.

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 `PSIM`

`PSIM` is an interconnection network simulator written in PCP, a parallel extension of C that supports efficient nested forking within programs [Brooks III 1988]. `PSIM` is a fork-join program with two parallel phases per iteration, where 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 described in Section 3.4. The work per task (loop iteration) is much larger in some loops than in others.[4] Nevertheless, the load is well balanced when P is *even* because the two largest loops are executed by different sets of processors. However, because of the processor-splitting between loops, significant performance degradation occurs when P is *odd* since half of the processors have to accomplish a larger amount of work in this case.

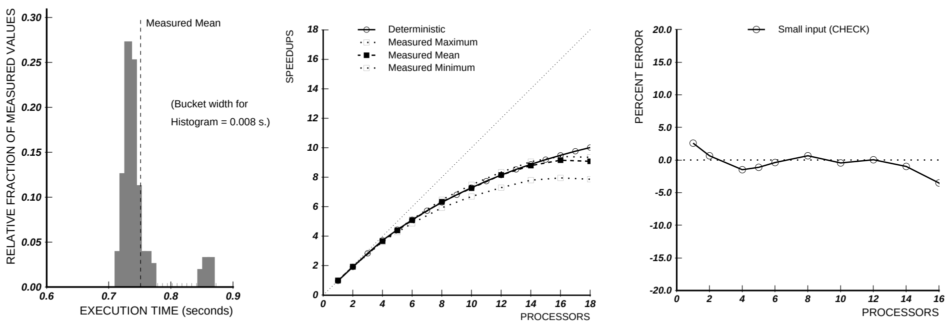
We consider two input networks for `PSIM`, containing 1024 and 4096 processors respectively. The percentage errors in the predictions for the two input sizes are shown in Figure 5 (b), and the measured and predicted speedups for the larger input (including the predicted speedup ignoring communication overhead) are shown in Figure 5 (a).

The 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 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 relatively high communication overhead due to bus contention (e.g., bus utilization is 0.81 for the larger input size on 16 processors), and this too is accurately captured by the predicted mean communication costs due to contention, as can be seen from Figure 5 (a). Despite the high communication and contention, ignoring the variance in communication delays and task times again appears to have little impact on model accuracy.

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 consisting of two iterations, with each iteration ending in a barrier (Figure 1(c)). The computation is partitioned into 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 we consider, most tasks require less than 10 milliseconds of execution time while a few require 100 milliseconds or more. Two scheduling options 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 (this

[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) Histogram of execution times in different runs

(b) Predicted and measured speedups

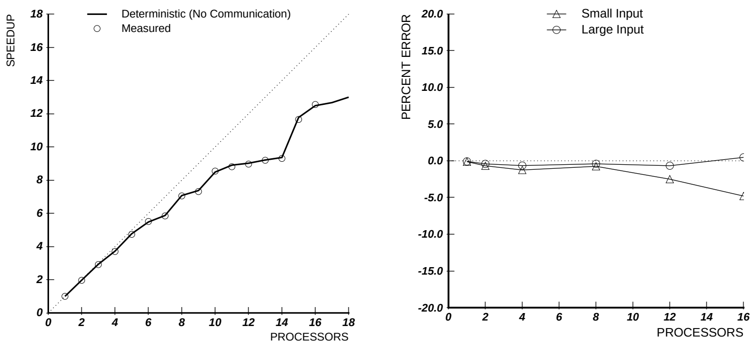
(c) Error in predicted execution times

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

is directly supported in our model implementation). The other option is a semi-static scheduling method in which geographic areas of the VLSI chip are allocated to specific processors. We will use our model to study the performance of this scheduling option and variants thereof in Section 5.3.

An important characteristic of `LocusRoute` is that the CPU requirements of the tasks in `LocusRoute` can vary from one execution to the next [Adve 1993]. This happens 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 of `LocusRoute` to vary significantly from one execution to the next, for the same input. This is illustrated in Figure 6 (a), which gives 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 predictions against the *range* of measured speedups, i.e., the minimum and maximum, as well as the mean. All speedup values are computed relative to the *mean measured* execution time on 1 processor. The range as well as the mean of the measured values in the following experiments were obtained from 40 runs of the program for each number of processors.

Figure 6 (b) compares the speedup predicted by the deterministic model with the range and the mean of the measured speedups. Figure 6 (c) shows the error in the predicted speedup relative to the mean measured speedup values (note that the measured execution times are mostly clustered near the mean and therefore comparisons against the mean are meaningful at least for this input, although the more complete picture provided by Figure 6 should not be ignored). The deterministic task graph model is quite accurate for this program compared to the mean measured values. An interesting feature of this example 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 deterministic model demonstrates that



(a) Speedups for large input (b) Error in predicted execution time

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

the model has successfully captured the impact of the order of execution of tasks, in addition to the precise allocation of tasks to processes.

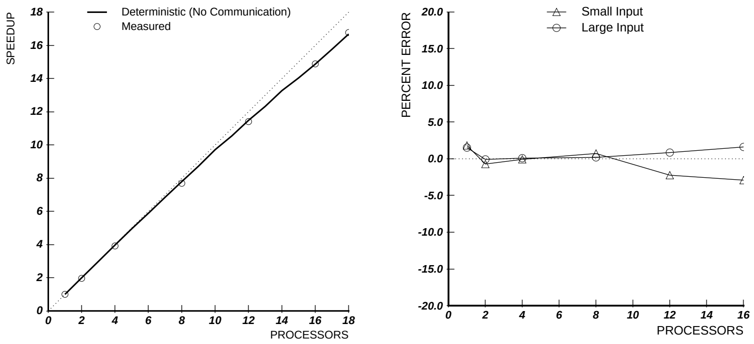
Figure 6 (b,c) also show that the error relative to either the mean or the most distant measured value shows an increasing trend at higher values of P. Additional detailed measurements showed that the individual task CPU requirements (excluding communication costs) on 8 and 16 processors were significantly 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 the model, the predicted execution times for $P = 8$ and $P = 16$ 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.

In general, when analyzing a program with variable CPU times, it will be necessary to informally understand or formally quantify the variability in execution times, before being able to draw broader conclusions about program performance from model predictions. For `LocusRoute` in particular, our model predictions are sufficiently accurate (both qualitatively and quantitatively) for exploring program performance issues. We use the model in Section 5.3 to evaluate various interesting changes to this program.

Results for `Polyroots`

The remaining two programs, `Polyroots` and `DynProg`, have non-series-parallel task graphs and (as noted in Section 4.2) the basic model suffices for both programs. `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). The tasks are dynamically scheduled using a single task queue, as in `LocusRoute`.

The percentage error in model predictions for the two input sizes (input polynomials of degrees 20 and 30) for program `Polyroots` are shown in Figure 7 (b). For the larger input size, the predicted execution times are all *within 1%* of the measured values. With the smaller input size, the errors are slightly higher because



(a) Speedups for large input (b) Error in predicted execution time

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

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, but 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 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).

4.4 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. This scheduling method was specified explicitly as an input to the deterministic task graph model.

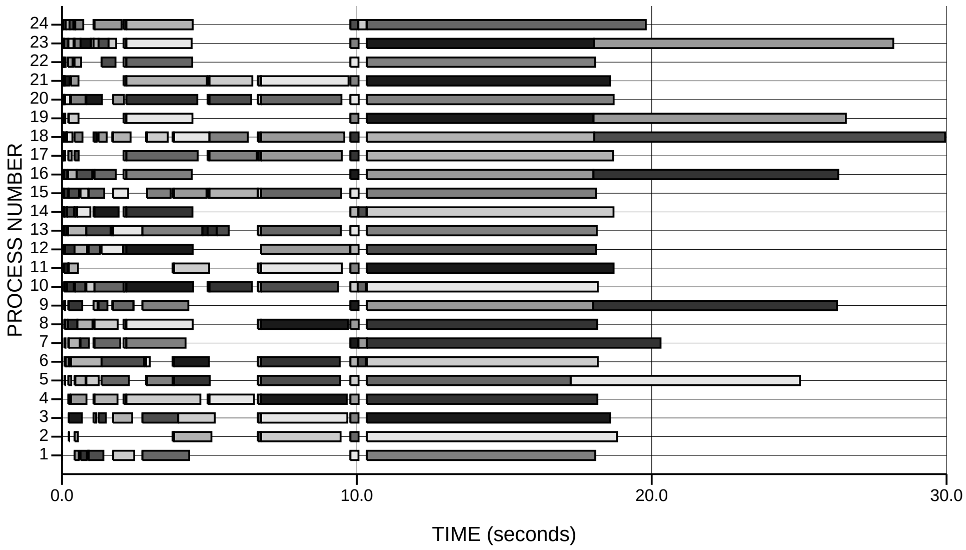


Fig. 9. Process timelines from deterministic model showing individual task executions for `Polyroots`. Input polynomial degree 30: $N = 348$, $Pmax = 30$, $P = 24$.

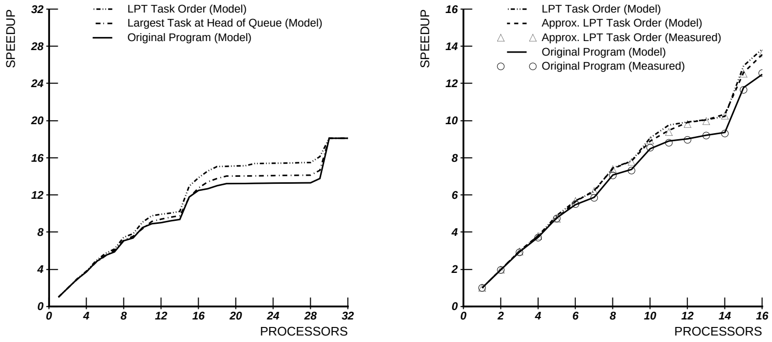
For `DynProg`, we again used two input sizes, namely G=100 and G=500. The percentage errors in the predicted execution time for the two input sizes are shown in Figure 8 (b) and the predicted and measured speedups for the larger input are shown in Figure 8 (a). Even though the absolute execution times 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 (in Section 2) that the deterministic task graph model can be useful for evaluating program design issues. In this section, we illustrate this point by evaluating 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 implementation) 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, Figure 9 shows time-lines of execution of each process computed from intermediate results of our model. A continuous band of a fixed shade represents the execution



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

Fig. 10. Program **Polyroots**: the effect of reordering tasks in final Phase

of a single task by the corresponding process, and an empty interval indicates the process is idle. This data provides 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.

The last two observations 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]. The model can easily be used to *predict* the effect of these two changes merely by specifying the new scheduling functions. (For our model implementation, this just requires reordering the task CPU requirements in the model input.) The predicted speedups with the two heuristics are compared to the original program in Figure 10(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 the tasks in the LPT order would yield a more significant improvement in speedup over a wide range of P, indicating that it could be worthwhile to attempt to implement the LPT heuristic in the program.

In fact, the LPT order can be *approximated* in the program with very little additional computation at the start of the phase. Each task in this phase executes a binary search on an interval of the real line, so an approximate reordering can be achieved by sorting the tasks in decreasing order of interval lengths. We call this the Approximate LPT order. Figure 10(b) compares the measured speedups of this modified program for $1 \leq P \leq 16$ to the predicted speedup for LPT, as well as the predicted speedup for the Approximate LPT order, and to the original program. The figure shows that the simple approximation to LPT was able to realize almost the full improvement possible with LPT. More important in the

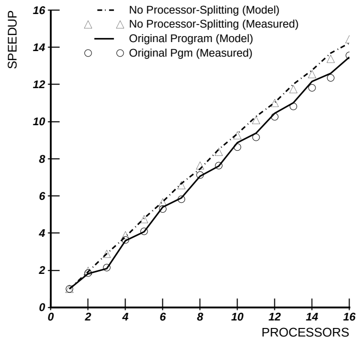


Fig. 11. Improvement in speedup of `PSIM` without splitting processors between loops

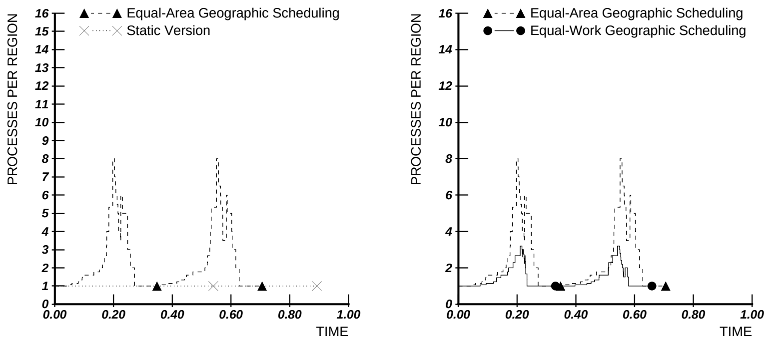
context of this work, the model was able to provide insight into a performance bottleneck, accurately predict the performance impact of the various modifications, and correctly predict that it would be worthwhile to attempt the full task reordering. This is possible because of the accurate representation 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`. We also implemented this change and measured the new execution times. In Figure 11, we compare the predicted and measured speedups for the original and modified task scheduling method. The model quickly and accurately predicts that the simpler static loop scheduling gives a 5-10% improvement in speedup as well as a smooth speedup curve. 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.

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



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

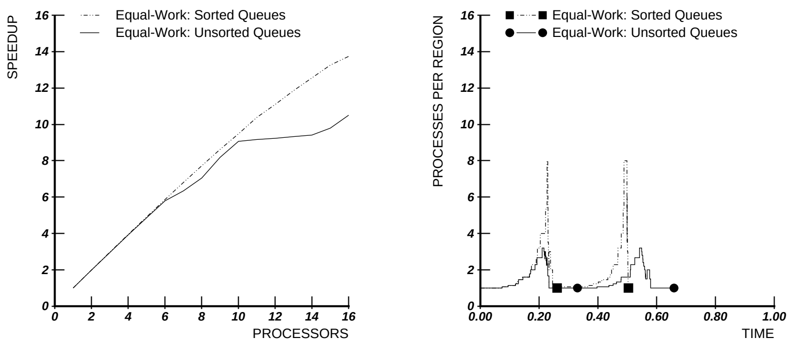
Fig. 12. Mean number of processes per region during an execution of `LocusRoute` ($P = 16$)

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 data locality is complex and difficult to predict because of the dynamic load balancing and because long wires can span multiple regions (requiring processes working on separate queues to communicate shared data). To gain some intuition about the trade-off between data locality and load balancing in `LocusRoute`, we use the model to compute the average number of *active processes per active region* as a function of time during an execution of the program. This metric gives a qualitative view of data locality, in that a lower value indicates that relatively fewer processes are sharing and updating shared data.

Figure 12(a) plots this average as a function of time in an execution on 16 processors, using 16 regions for semi-static scheduling. We also use the model to plot a hypothetical static version that has the same initial allocation as the semi-static, 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 12(b). The



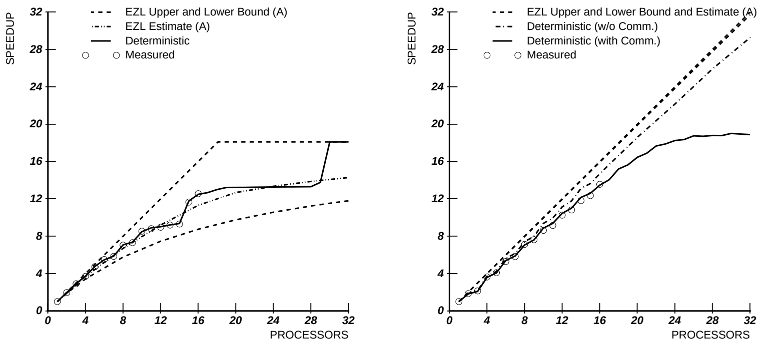
(a) Predicted speedups (b) Processes per region during an execution

Fig. 13. Predicted impact of reordering tasks with balanced semi-static scheduling in `LocusRoute` ($P = 16$)

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.

Figure 12(b) also shows that in either geographic scheduling option, a single large task is active at the end of each iteration. This indicates that (in addition to the equal-work geographic allocation) further performance improvement might be obtained by initially sorting the queues using the LPT heuristic. As in `Polyroots`, we can use the model to estimate the impact of this change. First, ignoring the effect of this change on locality, the model predicts that this change has the *potential* to significantly improve speedup for $P > 10$, as shown in Figure 13(a). Analyzing the precise impact of this change 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 13(b). As would be 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 while improving load balancing, but the improvement has sufficient performance potential to 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. Although these results are specific to `LocusRoute`, the various scheduling options that use multiple task queues are representative of typical methods for improving communication locality while preserving acceptable load balance in other programs. Thus, similar experiments are possible for other such programs as well.



(a) `Polyroots`: $N = 348$, $A = 18.1$ (b) `PSIM`: $N = 40963$, $A = 3407$

Fig. 14. Bounds using Average Parallelism computed from the Deterministic model

5.4 Computing Abstract Parallelism Metrics

Previous studies have shown that a few key parameters such as the fraction of sequential work [Amdahl 1967], average parallelism [Eager et al. 1989], and variance of parallelism [Sevcik 1989] can each provide a concise yet powerful characterization of program performance. For example, Eager et al. have derived bounds on speedup for arbitrary, work-conserving, task scheduling functions using the average parallelism (A) alone [Eager et al. 1989]. Sevcik has shown that for scheduling parallel jobs on a multiprocessor system, where only very concise information about the arriving jobs can be used, effective scheduling is possible using just two parameters, average parallelism and variance of parallelism [Sevcik 1989].

While simple parameters like the fraction of sequential work and $P_{\max}$ can be easily estimated for a given program, other parameters like A and the variance of parallelism are much more difficult to obtain. (For example, consider estimating A for `PSIM` or `Polyroots` in Figure 1(b,d).) Direct measurement of A (the speedup on $P = P_{\max}$ processors) is impractical for most scientific and engineering applications where $P_{\max}$ can be very high. [5] On the other hand, we can compute A analytically for a given task graph with a single solution of the basic model at $P = P_{\max}$: $A = T(1)/T(\infty) = \sum_{i=1}^{i=N} T_i/T(P_{\max})$, where $T(P)$ denotes the total execution time on P processors.

We used this method to compute A for the programs $POLY$ and $PSIM$, and the speedup bounds using these values of A are shown in Figure 14. Each figure shows the speedup bounds based on A, the speedup estimate obtained by interpolating those bounds, the speedup estimate from the deterministic task graph model, and the actual measured speedups. The figure for $PSIM$ also shows the speedup estimate from the basic deterministic model ignoring communication. For `Polyroots`, the speedup bounds are valid and the speedup estimate based on the bounds is fairly accurate, although it does not show the non-uniform behavior of the speedup curve, apparent from the deterministic model. `PSIM`, however, violates both basic

[5] A trace-driven technique developed by Larus for studying the *available loop-level parallelism in sequential programs* can, in fact, be viewed as computing the value of A that would be obtained if the program loops were parallelized [Larus 1993].

assumptions used to derive the bounds: it uses static scheduling of tasks which is not work-conserving, and it has significant bus contention causing communication cost to increase as the number of processors increases. Furthermore, since $P \ll A$ for $PSIM$ ($A = 3407$ for the large input), as also for many other scientific and engineering applications, the $\langle \mathrm{A} \rangle$ bounds provide little information. As shown by the Deterministic model, however, the static scheduling fails to achieve the full available parallelism, and communication overhead has an additional negative impact that increases significantly with P.

5.5 Summary

To summarize the results of this section, we used the deterministic task graph model to predict the performance impact of program design changes that affect load-balancing in two programs, and to explore design changes in another program that affect load-balancing as well as communication locality. The former experiments show the ability of the model to precisely represent the important details of task scheduling and to accurately compute synchronization costs. The latter experiments show that the model can also be used to obtain task and process level information that can be useful for comparing design issues that affect locality as well as load-balancing. We also showed that the basic deterministic model can be used to compute simple but fundamental metrics that characterize program parallelism and its impact on program performance. These metrics provide interesting insights into available parallelism for some programs, whereas the task graph and associated parameters provide far more detailed quantitative information about parallelism, scheduling, and communication. Overall, we believe these results indicate that the model can support useful and potentially important performance prediction for parallel programs.

6. COMPARISON WITH RELATED WORK

In previous work [Adve 1993], we have 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; Frank et al. 1997]. These models are primarily useful for obtaining broad, qualitative insights and bounds on program performance and scalability. Our work is complementary, and is aimed at much more detailed performance analysis and prediction, based on detailed model inputs. For 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 NW and perhaps $PSIM$ could be written in this form.

Finally, two recent 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.

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; Crovella et al. 1995]. In general, these are all focused on message-passing systems and static scheduling disciplines. 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]. [6] Their 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.

van Gemund [van Gemund 1993; van Gemund 1996] describes a modeling language (Pamela) for parallel programs that provides simple, inexpensive analysis of task scheduling and resource contention. His analysis, however, is restricted to series-parallel task graphs, to simple static or work-conserving dynamic task scheduling, and uses very simplistic bounds for computing communication costs with resource contention. His analysis has been validated only against synthetic series-parallel task graphs.

Models Restricted to Fork-Join Programs

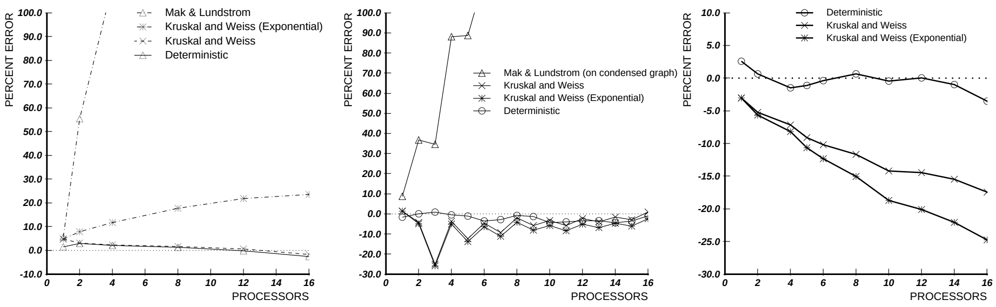
There a number of previous models restricted to programs with *fork-join task graphs* [Dubois and Briggs 1982; Heidelberger and Trivedi 1983; Towsley et al. 1990; Ammar et al. 1990; Kruskal and Weiss 1985; Vrsalovic et al. 1988; Cvetanovic 1987; Tsuei and Vernon 1990], divide-and-conquer task graphs [Madala and Sinclair 1991] or pipelined task graphs [Lewandowski et al. 1996]. 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 program with 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 validate the model for a number of task time distributions, and 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. Such an MVA framework and analytical parameter estimates could be used directly in combination with our high-level model as well.

6.2 Quantitative Comparisons with the Deterministic Task Graph Model

As mentioned earlier, we have evaluated several representative stochastic models for the same applications used in Sections 4 and 5 [Adve 1993]. None of these models had been evaluated previously using real programs. The primary goal of our comparison was to bring out the key advantages and disadvantages of assuming

[6] The results in Section 6.2 show that the ML model solution is too expensive to permit modeling tasks of low task time variance by a sequence of exponential tasks.



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

Fig. 15. Comparisons between our Deterministic model and previous stochastic models

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] (using the simplest or the most general known model in each case). 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 which 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 cost proved very high, and except for the smallest cases, the only way we were able to apply the ML model was to use the condensed task graph. (This is only practical for statically scheduled programs, and we could not apply the ML model to `LocusRoute` as well.) 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 15. 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.

Summary. In summary, the above results show that simple stochastic models based on the mean and variance of i.i.d. task times (including [Kruskal and Weiss 1985; Madala and Sinclair 1991; Ammar et al. 1990]) are accurate for programs with simple synchronization structures and restricted task scheduling, but cannot capture specific details such as task ordering. Nevertheless, these models are attractive because they can be applied when detailed individual task times are not available, whereas the more detailed models (including ours) cannot. The more general models based on exponentially distributed task times (such as [Mak and Lundstrom 1990; Thomasian and Bay 1986; Mohan 1984; Kapelnikov et al. 1989]) appear too inefficient to use even for relatively small task graphs, and can yield large errors with the condensed task graph.

In contrast, Section 4 shows that our model is accurate, efficient, and generally applicable to a wide class of real programs. The approximation of deterministic task times is key to achieving these goals because it greatly simplifies the calculation of synchronization delays and permits detailed modeling of task scheduling, without significantly compromising model accuracy.

7. CONCLUSIONS

In this paper, we have proposed and validated an analytical model for parallel program performance prediction, and used several examples to illustrate that the model can be used to understand and improve the performance of real programs. The model we propose is applicable to programs with arbitrary static 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, insight obtained using the model suggested design changes to improve load-balancing and accurately predicted the performance impact of the design changes. In a further program, novel detailed metrics from the model were used to obtain insight into and explore the impact of design changes that improve communication locality as well as load-balancing.

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. While several previous analytical models have been based on similar task graph representations, all such models have assumed stochastic task execution times. The key advantage of the deterministic assumption is that it implies a unique task execution sequence for the program. Therefore, the model permits an efficient and straightforward solution based on critical path analysis, modified to account for task scheduling precisely. Furthermore, the model is accurate because it accurately represents key details of task scheduling, the order of task execution, non-uniform task times, and average communication costs. Compared with previous models applicable to a general class of programs, we showed that our model is more accurate and much more efficient. Compared with previous models applicable to simple task graphs, our model requires a much more detailed input program description but applies to a wider class of programs and provides more detailed and more accurate characterization of program performance.

The experiments and results in this paper also suggest several possibilities for future research. The primary limitation of our model, shared by previous program models, is that it does not provide the means to predict how the communication parameters for the tasks (e.g., cache miss rates) vary for different systems and task scheduling methods. To fully exploit the predictive capabilties of the model, we believe the key remaining challenge is to be able to predict how the communication parameters vary using some *intrinsic description of communication behavior* (just as our model is able to predict the impact of synchronization and task scheduling from the task graph which is an intrinsic description of the parallelism structure). In a different direction, 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. 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. S. 1993. Analyzing the Behavior and Performance of Parallel Programs. Ph.D. thesis, University of Wisconsin-Madison. Available as UW CS Tech. Rep. No. 1201, or via `http://www.cs.rice.edu/~adve/publications.html`.

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.

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 International Conference 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 Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA.

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

CROVELLA, M. E., LEBLANC, T. J., AND MEIRA, W. 1995. Parallel Performance Prediction Using the Lost Cycles Toolkit. Tech. rep., Department of Computer Science, University of 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. Fifth ACM SIGPLAN Symposium 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 (April), 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*.

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, University of Vienna, Institute for Software Technology and Parallel Systems.

FAHRINGER, T. AND ZIMA, H. 1993. A Static Parameter-Based Performance Prediction Tool for Parallel Programs. In *Proceedings of the 1993 ACM International Conference 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 *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, USA.

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 *Proceedings of 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.* Rockville, Maryland.

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.

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.*

LEWANDOWSKI, G., CONDON, A., AND BACH, E. 1996. Asynchronous Analysis of Parallel Dynamic Programming Algorithms. *IEEE Transactions on Parallel and Distributed Systems 7,* 4 (April), 425–438.

MADALA, S. AND SINCLAIR, J. B. 1991. Performance of Synchronous Parallel Algorithms with Regular Structures. *IEEE Trans. on Parallel and Distributed Systems 2,* 1 (January), 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.

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 Symposium on Parallel Algorithms and Architectures*, pp. 178–187.

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.

SEVCIK, K. C. 1989. Characterizations of Parallelism in Applications and Their Use in Scheduling. In *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 171–180.

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.

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. on Parallel and Distributed Systems 1,* 3 (July).

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 International Conference on Parallel Processing*, pp. II 185–189.

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 (November), 712–727.

VAN GEMUND, A. J. 1993. Performance Prediction of Parallel Processing Systems: The PAMELA Methodlogy. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo.

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

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 Symposium 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.

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.