Analyzing the Behavior and Performance of Parallel Programs

Vikram S. Adve

Technical Report #1201

December 1993

Analyzing the Behavior and Performance of Parallel Programs

Vikram S. Adve

Computer Sciences Technical Report #1201

University of Wisconsin-Madison

December 1993

† A postscript version of this technical report is available via anonymous ftp from ftp.cs.wisc.edu.

ANALYZING THE BEHAVIOR AND PERFORMANCE OF PARALLEL PROGRAMS

by

VIKRAM SADANAND ADVE

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1993

Abstract

An analytical performance model for parallel programs can provide qualitative insight as well as efficient quantitative evaluation and prediction of parallel program performance. While stochastic models for parallel programs can represent execution time variance due to communication and resource contention delays, a qualitative assessment of previous models shows that the stochastic assumption makes it extremely difficult to compute synchronization costs and overall execution times.

This thesis first re-evaluates the need for the stochastic assumption by examining the influence of non-deterministic communication and resource contention delays on execution times in parallel programs. An analytical model of program behavior, combined with detailed program measurements, provides compelling evidence that in shared-memory programs on current systems as well as programs with similar granularity on foreseeable future systems, such delays introduce extremely low variance into the execution time of each process between synchronization points, even with high communication costs and contention.

Motivated by the above results, the thesis develops a conceptually simple *deterministic* model for parallel program performance prediction, using deterministic values to represent mean task times including communication, and (if necessary) shared-resource contention computed from a separate, stochastic model. Experiments applying the model to several shared-memory programs demonstrate the efficiency, accuracy and ability to model programs with large and complex task graphs. A quantitative assessment of previous stochastic models shows that they have inconsistent or poor accuracy, as well as prohibitive computational cost in models applicable to complex task graphs. Furthermore, in comparison with simple, insightful speedup bounds computed using parameters such as average parallelism, the deterministic model provides additional qualitative as well as quantitative information, for comparable effort.

The thesis then uses example programs to demonstrate the insight and predictive power provided by the deterministic model. The model can be used to quantify and understand nuances of

\20\

program performance, and to quickly predict the impact of system changes as well as program design changes that affect load-balancing, such as changes in the partitioning and scheduling of tasks. This insight and predictive power is due to the particular task-graph-based representation of program parallelism and scheduling.

In summary, the analytical and experimental results in the thesis contribute towards understanding a fundamental principle of parallel program behavior, and towards evaluating, understanding, and predicting parallel program performance.

Acknowledgement

During the course of my Ph.D., many people, far too numerous to mention here, have given help, advice, suggestions, support, and friendship. A few have made special contributions to my thesis work, my learning and research experience, or in more personal ways. It is a pleasure to acknowledge them here.

Mary Vernon, my thesis advisor, stimulated my interest in performance modeling and evaluation, and guided my ideas through numerous discussions. Over the years, she has provided invaluable guidance, technically and professionally. For her support, criticism, advice and encouragement, I am truly grateful.

My experience in analytical modeling has been enriched by my interaction with Randy Nelson, who gave me a deeper appreciation for the power and elegance of mathematical models. Mark Hill helped to greatly enhance my long-standing interest in computer architecture. Anne Condon, Mark Hill, and David Wood provided valuable suggestions and comments on my thesis research.

I would like to thank my fellow students, Amarnath Mukherjee and Rajesh Mansharamani, for our numerous, stimulating discussions and for their technical advice and encouragement.

It is a special pleasure to thank my parents, and Sarita's parents, for their immeasurable support in our work, and their sense of involvement in our failures and successes. I would also like to thank my grandfather for introducing me to the crossword and to P. G. Wodehouse, unfailing outlets from the "irksome captivity" of classes and research.

My profound gratitude goes to my father who strongly encouraged me to pursue basic research, and whose dedication to his work will be a continuing source of inspiration to me.

Finally, my wife, Sarita, has been colleague and companion, best friend and severest critic. Graduate school has been made so much less tedious by her companionship during many long working hours, and life outside computer science so much more enjoyable by her company.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Developing programs to extract the available processing power of a parallel computer remains much more difficult than doing so on a high-performance sequential computer. New techniques to simplify the task of parallel programming will be necessary before truly general-purpose parallel computing becomes prevalent. Important among these will be performance evaluation techniques that can help a programmer to understand the key qualitative aspects of parallel processing and to obtain quantitative estimates of program performance.

To assist in effective parallel programming, performance evaluation techniques must address three basic needs. The first is to provide an understanding of the fundamental principles of parallel program behavior and their impact on program performance. The second is to enable the programmer to evaluate the performance of a particular program on a particular system and thus obtain insights that can suggest potential improvements in the program. The third is to enable the programmer to predict the impact (on program performance) of design changes in the program or changes to the underlying system or system configuration.

Performance evaluation tools in use today for evaluating parallel program performance are based on measurement or simulation. Measurement-based performance analysis tools such as Pablo [RAM92], IPS-2 [MCH90] and numerous others provide the ability to evaluate the performance of a given program on an existing system in detail. Simulation-based tools such as the Rice Parallel Processing Testbed [CMM88], the Wisconsin Wind Tunnel [RHL93], and others provide the additional flexibility of evaluating an existing program on varying system sizes and configurations. Thus, these techniques collectively achieve at least partial success in addressing the second and third of the three goals of parallel program performance evaluation stated above.

An underlying thesis of this dissertation is that analytical performance models can make important new contributions towards meeting each of the above three basic goals. The ability of analytical models to provide basic principles about various aspects of system behavior and performance is well-established; for parallel programming, this ability can be exploited to derive fundamental principles of program behavior and their impact on performance. (A simple and well-known example of an analytical model fulfilling such a role is Amdahl's Law [Amd67].) Furthermore, because analytical models have the ability to view a program and the underlying system at a higher level of abstraction than measurement or simulation techniques, they can play an important complementary role to measurement and simulation tools in meeting the second and third of the goals above. The more abstract representation of a parallel program can help to provide new insights into, as well as efficient quantitative estimates of, program performance. The abstract representation of program and system behavior in such models also makes it possible to analyze hypothetical programs and systems as well as combinations of these, thus providing the predictive power required to address the third goal above. The research described in this dissertation provides evidence of the ability of analytical models to fulfill each of these three roles.

The remainder of this chapter motivates and states the specific problems studied in this dissertation (Section 1.1), lists the contributions of the work (Section 1.2), and finally gives a detailed outline of the remainder of the dissertation (Section 1.3).

1.1. Motivation and Statement of Problems Studied

The principal factors affecting the performance of a parallel program are the computational work that must be accomplished, the communication required between processes, contention for shared hardware and software resources during computation and communication, and synchronization delays experienced when one process must wait for another process to reach some point in its execution.

An important influence on analytical models so far has been the non-deterministic nature of some of the above factors. In particular, inter-process communication events and contention for shared resources introduce non-deterministic delays into the execution of each process of a parallel program. Furthermore, some programs also have non-deterministic processing requirements - that is, the CPU requirements of the program vary significantly across different executions on a particular input. Performance models of parallel programs and systems have typically used stochastic task execution times to represent non-determinism due to these various sources. In fact, most previous analytical models for parallel programs are stochastic models [AIA91, DuB82, HaM92, KME89, KrW85, LCB92, MaS91, MaL90, ThB86].

Non-deterministic delays due to communication and resource contention increase the variability of process execution times (besides increasing the mean execution times), which in turn affects synchronization costs. In stochastic models, this is reflected by taking into account the variance or distribution of execution time of the synchronizing processes in the program when computing the expected delays at synchronization points. However, estimating average synchronization delays in stochastic models can be extremely difficult, except for programs with very simple synchronization behavior. In fact, reviewing the assumptions and solutions techniques of previous analytical models in Chapter 2, we find that many previous models apply only to programs with extremely simple synchronization structures [AIA91, Cve87, DuB82, KrW85, LCB92, MaS91, TsV90, VSS88], while other models require complex and heuristic solution techniques to model programs with more sophisticated structures [KME89, MaL90, ThB86]. Furthermore, models in the latter class have to assume exponentially distributed task execution times to permit tractable solutions. The exponential assumption, however, implies a high variability in execution time and no data is available showing the effect of this assumption on the accuracy of the models. (To our knowledge, none of the models in this class has previously been tested using actual programs.)

The above observations suggest that it is important to re-evaluate the assumption of non-deterministic task times. In particular, in many parallel programs, the CPU requirements *are* fixed or almost fixed for any particular input. (Some evidence for this is provided in the thesis.) For such programs, does the variability due to random delays justify the stochastic assumptions in these performance models? More generally, how do random delays influence the variance of execution time of synchronizing processes in parallel programs? To our knowledge, these questions have not previously been addressed. Furthermore, these are questions about the fundamental behavior of parallel programs, and as such should yield principles about program behavior that would be useful not only for performance evaluation, but for programmers as well. As argued above, an analytical modeling approach could be well-suited to addressing such questions. Thus, one goal of this thesis research is to develop and use an analytical model of program behavior to study the influence of random delays on execution times in parallel programs.

For analytical performance techniques to be able to address the second and third of the basic needs mentioned above, an efficient, accurate and practical analytical model for evaluating program performance is necessary. The second major goal of this thesis research is to develop an analytical model for parallel program performance prediction, demonstrate that it meets these criteria, and provide evidence that it can be used to obtain insight into program performance and to predict the performance impact of program and/or system design changes.

Although numerous analytical models for parallel program performance prediction have previously been developed as mentioned above [AIA90, Cve87, DuB82, HaM92, KME89, KrW85, LCB92, MaS91, MaL90, ThB86, TsV90, VSS88], the efficiency, accuracy and practical usefulness of these models has remained an open question. In fact, in many of these cases, *no data* has been provided describing model accuracy or efficiency. Thus, there is no quantitative evidence by which to evaluate the strengths and limitations of the various models, and by which to understand the state of the art.

The rationale for the approach we take in developing a model is contained in our qualitative assessment of previous stochastic models, along with the results of the study of the influence of random delays. That is, previous stochastic models have required complex heuristics and the exponential task assumption to analyze programs with other than the simplest task graph structures, whereas our study of the influence of random delays indicates that the exponential assumption represents much higher variance than found in practice in many programs. In fact, a key implication of that study is that it could be reasonable to *ignore* the variance of task and process execution times when computing synchronization costs in a parallel program. (The intuition and specific results leading to these conclusions will be described in detail in a subsequent chapter.) These results suggest that it is worthwhile to explore the use of a *deterministic* model for addressing the second major goal of the thesis, stated above. In fact, two previous deterministic models have been developed for parallel program performance prediction, but as discussed in Chapter 2, both have been restricted to programs with very simple structures and with limited or no ability to represent synchronization costs or task scheduling.

Motivated by the above arguments, the second half of this thesis develops and validates a deterministic model for parallel program performance prediction, testing the accuracy, efficiency and practicality of the model for real parallel programs on realistic input sets. As part of this study, the model is compared with previous stochastic models, thus obtaining quantitative results on the accuracy and efficiency of these models to provide some understanding of the state of the art. The model is also compared with parametric speedup bounding techniques that can be used to obtain qualitative insight into parallel program performance. Finally, the deterministic model is used to understand the performance of existing programs, to evaluate program performance on hypothetical larger systems, and to evaluate the performance impact of hypothetical design changes in these programs. These experiments, together with the analytical model used to study the influence of random delays, support our underlying thesis stated at the outset, namely that analytical models can contribute towards meeting each of the three basic requirements of parallel program performance evaluation.

1.2. Contributions of the Thesis

The principal contributions of this thesis are as follows.

- We develop an analytical model of process behavior in a parallel program in the presence of random delays. The model, based on a simple renewal process representation, yields considerable insight into the effect of random delays on the variance and distribution of process execution time over any interval of execution.
- Using detailed program measurements to parameterize and apply the model, we provide compelling evidence that in current shared-memory programs, as well as programs with similar granularity on foreseeable future systems, communication delays introduce very little variance into the execution time of a process between successive synchronization points, even under conditions of high communication cost and contention.
- We use direct measurements of program execution time to show that, for many but not all such programs, processing requirements also introduce very little variance into the execution time between synchronization points.
- We describe a *deterministic* model for parallel program performance evaluation. The deterministic assumption enables a conceptually simple, computationally efficient, solution, while the carefully defined, abstract task-graph-based representation of program parallelism provides significant flexibility and predictive power for studying important program design issues. The model applies to parallel programs with arbitrary task graphs and a wide class of task scheduling disciplines.
- We show that the deterministic model is accurate, with typical errors in the running time estimate of less than 10% in all cases tested, and very often only about 2-3%. The model is also extremely efficient in practice, and can easily be used for programs with tens of thousands of tasks. Two programs we studied cannot be evaluated in practice by any previous analytical

model we are aware of.

- We provide data to show that of previous stochastic models, the simplest models, which are restricted to programs with simple (fork-join) synchronization structures, can be accurate for programs that satisfy further simplifying model assumptions concerning task times and scheduling, but can have significant errors otherwise. For the more general models, the assumption of exponentially distributed task times often leads to poor accuracy. The computational requirement of such models is also extremely high, precluding analysis even of programs with fairly small task graphs.
- Comparing the deterministic model with the parametric speedup bounds of Eager, Zahorjan and Lazowska [EZL89], we show that for essentially the same effort as that required to calculate parameters for the bounds, the deterministic model can be used to obtain estimates of the actual speedup. Besides being quantitatively much more accurate, these estimates provide important *qualitative* information about program performance not available from the bounds. Furthermore, the deterministic model applies to a large class of programs not covered by the bounds, such as programs with static task scheduling.
- We provide evidence to demonstrate the insight and predictive power available with the deterministic model, showing examples where the model was used to evaluate the performance of existing parallel programs on hypothetical larger systems, as well as to evaluate the performance impact of program design changes that affect load-balancing in a program, such as changes in the partitioning and scheduling of tasks.

1.3. Outline of the Thesis

This thesis is organized as follows. Chapter 2 first defines key terms and concepts as they will be used throughout this work. The bulk of the chapter reviews previous analytical models for parallel program performance prediction, describing the previous models in a common hierarchical framework that is helpful in understanding the important features of the models as well as the principal difficulties of the general problem. This chapter also reviews relevant details of the parametric speedup bounding techniques.(A few previous arguments and conjectures by previous authors that have bearing on the study of the influence of random delays are described in Chapter 3.)

Chapter 3 describes the analytical model and experimental results of our study of the influence of random delays on synchronization costs in parallel programs. The implications of these results for parallel program performance prediction are discussed in the following chapter. Other implications of the results are discussed and illustrated in Chapter 8.

Motivated by the above results, Chapter 4 develops a conceptually simple deterministic model for parallel program performance prediction, and discusses issues that arise in implementing and using the model.

Chapter 5 uses five shared-memory programs to evaluate the accuracy, efficiency and general applicability of the deterministic model. Three of these programs are used to evaluate the efficiency and accuracy of representative stochastic models as well, and thus also to compare the deterministic and stochastic models.

Chapter 6 compares the deterministic model with the parametric speedup bounds of Eager, Zahorjan and Lazowska [EZL89], in terms of the complexity of applying the two techniques for specific programs and the qualitative and quantitative information provided by the two techniques.

Chapter 7 discusses the examples where the deterministic model was used to predict and understand the performance of programs on existing and larger machines, and to evaluate design trade-offs in the programs.

Chapter 8 briefly discusses other implications of the results obtained from the study of random delays.

Finally, Chapter 9 summarizes the conclusions of the research reported herein, and explores several related questions that could be addressed in future research.

Chapter 2

Preliminaries and Previous Work

To ensure consistent terminology as well as to provide a perspective on this work, we begin by defining several key terms and concepts in Table 2.1. The fundamental concepts are those of *task* and *task graph.* (Our definitions of these terms are similar to the sub-task and the graph model of parallel software used in [EZL89].)

A set of tasks for a program on a particular input represent inherent units of computation, because of two properties conferred by the definition: (1) A task is executed sequentially, i.e., by a single process, in any execution of the program, and (2) A precedence constraint between a pair of tasks can only occur at a task boundary.[1] A task graph describes the parallelism structure of a computation. In many programs, the task graph is the same for all executions on a particular input, regardless of the number of processes or processors used, and also regardless of the scheduling function used; in such programs, the task graph is a representation of the inherent parallelism structure of the program. For example, consider a parallel loop with N independent iterations, where each iteration is always executed by exactly one process. By our definitions, each iteration forms a task and the task graph for this loop consists of a single parallel phase with N tasks. Various scheduling functions are possible for such a loop (e.g., static scheduling in blocked or cyclic order, dynamic scheduling, guided self-scheduling [PoK87], etc.), but the task graph is the same in all cases.

In some programs, however, the task graph (or some portion thereof) may necessarily depend on the number of processes used during program execution. A common and simple example occurs when the individual processes require significant and unequal computation for initialization, which must be represented using one task per process in this portion of the graph. Even for such programs,

1. Note, however, that we are not assuming that the set of tasks representing a program on a particular input is unique. For any program and input, more than one possible set of tasks may meet the definition and thus possess these properties. For example, some tasks in one such set may be broken down into tasks with smaller granularities, yielding another set of tasks for the same program and input.

Table 2.1. 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 where tasks are statically allocated 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 [TRS90].

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 [HaM92].

This class includes fork-join graphs but excludes, for example, the task graphs in Figures 5.1 (d,e).

---

the task graph provides a useful (though less elegant) representation of the parallelism structure.

For the above reasons, we believe the task graph provides an appropriate level of abstraction for an analytical model, and we use it together with a task scheduling function as the basic input to our performance prediction model developed in Chapter 4. It is a less detailed representation than those typically used during measurement or simulation of parallel programs, and yet provides sufficient information for evaluating many important program performance issues. Furthermore, given some reasonable understanding of a program, we have found (e.g., in the experiments in Chapters 5-7) that it is often not difficult to construct the task graph.

This choice of input representation has some limitations. First, in some programs, the task graph or the scheduling function for a particular input depend in complex ways on specific input values rather than in simple fashion on aggregate parameters such as total input size. In such cases, constructing the task graph or scheduling function may require somewhat greater effort. A more serious limitation of the task graph as a model of the parallel structure occurs when the graph (i.e., the tasks executed and the dependences between them) can vary from one execution to another, *for the same input*. For example, programs that use branch-and-bound algorithms can exhibit such behavior. In such cases, the task graph only represents a specific execution. Thus, the results of a task-graph-based model will be specific to that particular execution, and care is required in interpreting or generalizing the results.

A much more compact graph representation of a program is the *condensed graph*, specifically a directed acyclic graph whose basic units are the work performed by individual processes between synchronization points, rather than the individual tasks. It follows from the definition that, for any program, the condensed task graph depends on the number of processes used to execute the program, and on the scheduling function used to allocate tasks to processes. For programs with static scheduling and homogeneous tasks (at least in intervals between synchronization points), the condensed graph can be directly constructed from the original graph. Furthermore, a model that can be applied using the original task graph will usually also be applicable using the condensed task graph. This is potentially useful because the condensed graph can be orders-of-magnitude smaller. For example, in a parallel loop with N iterations (tasks) executing on P processors, if the iterations are statically and equally divided among P processes, the corresponding condensed graph would have P vertices, compared to N in the task graph. However, the choice of input graph can strongly affect the accuracy of a stochastic model, as will be shown in Chapter 5.

The definitions in the table distinguish tasks, processes and processors, whereas most previous models discussed in this paper only refer to tasks and processors. We use the same terms when describing those models. Throughout, we use N to denote the number of tasks in a 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 previous models for parallel program performance prediction 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 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. Most of the difficulty in developing a model usually lies in making this computation tractable, particularly in stochastic models where synchronization costs can be extremely difficult to compute.

Individual task execution times are computed from the lower-level model component. This component represents system-level effects such as communication costs, interconnection network contention, etc., and is usually a queueing network model of the system. The solution of this model component must account for the effect of task precedences and scheduling. In particular, the task precedences together with the task scheduling function imply that only specific combinations of tasks can be simultaneously in execution. In some models therefore, the queueing network is solved for every possible distinct combination of tasks in execution, while in one model task overlap probabilities derived from the solution of the higher level component are used to incorporate the effect of task precedences. (The latter model thus requires an iterative solution.)

The high-level model component of a model usually plays the primary role in determining the overall accuracy, efficiency and modeling power. In particular, in almost all models, the high-level component determines the representation of task scheduling and execution, and process synchronization. Thus, it determines to which programs (i.e. which classes of task graphs) the model can be applied and (especially in stochastic models) also how efficiently the model can be solved. Most important, the stochastic assumptions in a model affect its accuracy chiefly by affecting the accuracy of synchronization costs computed in the high-level model. Of course, the accuracy of the low-level model component also has an effect on the accuracy of the overall model. Nevertheless, for any particular high-level model and any particular system various choices of low-level model may be possible, depending on the specific experimental environment as well as the desired accuracy of the model results. Thus, the features of the specific low-level model component used in each model are not of fundamental importance, and will only be discussed where necessary.

2.2. Models Applicable to Arbitrary Task Graphs

We begin our discussion of previous work with three models that apply to arbitrary task graphs, and which illustrate the above hierarchical framework as well as the principal difficulties of the general problem.

Thomasian and Bay [ThB86] developed a 2-level hierarchical model in which task residence times are assumed to be exponentially distributed. This assumption allows the task-level behavior of the program to be modeled as a Markov chain in which each state represents one possible combination of tasks in execution, and the state transitions correspond to task completions in the program.

Task scheduling can be precisely accounted for when constructing the chain. The Markov chain is their higher level model component. By adding a transition from the end state back to the start state in the chain, the resulting Markov chain can be solved for the steady-state probability distribution and thus the average program completion time. The transition rates between states in the chain are derived from the lower level model, which is a queueing network model of the shared resources in the system. The queueing network has to be solved once for each distinct state of the high level model. The number of states in the state space, and thus the model solution cost, grow combinatorially with the maximum parallelism in the program (i.e. $O(2^N)$ in the worst case). The authors show the model to be accurate for a task graph with $N = 6$ tasks, compared to simulations that also assume exponential task times.

Mohan [Moh84] earlier described a model equivalent to that of Thomasian and Bay, but used a stochastic simulation to find the average program completion time by sampling different execution paths, instead of analytically solving for the steady-state probability distribution of the Markov chain.

Kapelnikov, Muntz and Ercegovac [KME89] also propose a very similar hierarchical model for evaluating programs on distributed systems. Their model assumes a *computation control graph*, a program representation that is more general than a task graph in that alternative control-flow paths can be represented in the graph. This representation includes task graphs and condensed graphs as special cases. They assume that each node in the computation control graph has an exponentially distributed execution time. In their high-level model, they use a Markov chain solution for specific segments of the graph, and then use numerous complex heuristics to aggregate the individual solutions for segments in series and parallel. The low-level model is a queueing network representing system resources as well as certain synchronization constraints, and is solved once for each state of the Markov chain for each segment. They describe one example evaluating a computation control graph with 15 vertices and a maximum parallelism of 2, but only compare their results to simulations of the graph that also assume exponential task times. Otherwise, the model accuracy and efficiency have not been evaluated.

It is important to note that, for a particular input graph, the above three models use a common underlying Markov chain representation of task-level behavior, principally because of the common assumption of exponential task times. Thus, the high-level components of the three models differ only in the solution techniques used. This equivalence implies that the exponential task assumption will have very similar impact on the accuracy of all these models. Furthermore, this effect of the exponential task assumption will extend to other models with equivalent representations as well. We will exploit this similarity in our comparison of deterministic and stochastic models described in Chapter 5.

2.3. A Model for Series-Parallel Task Graphs

All the above models explicitly consider the individual combinations of tasks that can be simultaneously in execution (i.e., the detailed state space). Mak and Lundstrom develop a heuristic and fairly complex graph reduction technique as their higher-level model component, to avoid considering the individual program states [MaL90]. This heuristic, however, restricts their model to programs that have *series-parallel task graphs*. It also requires the assumption that task residence times are exponentially distributed.[2] Given the individual mean task residence times, the graph reduction technique computes the overall program execution time by computing the expected maximum of series or parallel groups of tasks. In order to simply use the expected maximum of task groups, they have to ignore task scheduling and processor contention during the graph reduction. These are instead included in the mean task residence times computed in the lower-level model. In the lower-level, individual mean task residence times are calculated using a closed product-form queueing network in which each task forms a separate customer class and all shared resources (including processors) are represented as queueing centers. To avoid having to solve this queueing network for every possible combination of tasks in execution, the calculation of queueing delays takes into account the average time that the executions of each pair of tasks overlap. These average overlap times are computed as part of the higher-level graph reduction, and thus an iteration is required between the two model components. The space and time complexity of the model solution are O(N^2) and O(N^3) respectively. The authors test the accuracy of the heuristic approximations required to solve the model by simulating hypothetical task graphs with exponential task times and by measuring a synthetic program explicitly written with geometrically distributed task times.

2. In their paper [MaL90], Mak and Lundstrom mention that instead of assuming exponentially distributed task times, an Erlang distribution can be used to match the actual variance of task time if desired. However, they only derive model equations for the exponential task case and it appears extremely difficult to extend key parts of their model to allow other distributions. Thus, we use their model exactly as derived in their paper, with exponential task residence times.

Note that there is a Markov chain underlying the Mak and Lundstrom model as well, since it also assumes exponentially distributed task times. In general, however, this Markov chain is not equivalent to those in the previous three models because the high level model (which determines the Markov chain) ignores the number of processors and the task scheduling, as explained above. However, if the number of processors is greater than the maximum parallelism in the input graph, this Markov Chain *would* be equivalent to those in the previous three models. For example, all four models would have equivalent underlying Markov chains when used with the condensed task graph as input; in this case, the task scheduling is incorporated when creating the condensed graph itself.

2.4. Models Restricted to Fork-Join Programs

A number of previous models are restricted to programs with *fork-join task graphs* [AIA91, Cve87, DuB82, HeT83, KrW85, TRS90, TsV90, VSS88], and are all much simpler than models described so far. Of these, perhaps the most general is the seminal model of Kruskal and Weiss [KrW85]. They consider a parallel program consisting of N independent parallel tasks executing on P processors, and make two simplifying assumptions about task behavior. They assume task execution times to be i.i.d. random variables with an IFR distribution with mean μ and variance σ.[3] They also assume that tasks wait in a common queue and as a processor becomes available it is allocated a fixed-size batch of K tasks (incurring a fixed overhead of h time units). This forms their high-level model, and they do not specify a low-level model component, i.e., they do not specify how μ and σ should be estimated. Under the above assumptions, they derive the following simple estimate for the total execution time: $\frac{N}{P}\mu + \frac{Nh}{PK} + \sigma\sqrt{2K}\log P$. This estimate is asymptotically exact as $P \rightarrow \infty$ and $N/P \rightarrow \infty$, but has been shown to be fairly accurate compared to simulations for small values of P, for a number of task time distributions.

The models of Vrsalovic et al. [VSS88], Cvetanovic [Cve87] and Tsuei and Vernon [TsV90] are the three deterministic models mentioned in Chapter 1. The models of Vrsalovic et al and Cvetanovic apply to iterative parallel programs in which the computational work as well as the

3. A distribution $F(t)$ is said to be IFR, or *Increasing Failure Rate*, if $F(0) = 0$ and if, for any $t_0 > 0$, $\frac{1-F(t+t_0)}{1-F(t)}$ is monotone increasing in t. IFR distributions are continuous and have decreasing mean residual life [Wol89]. For example, the Erlang and exponential are both IFR but the hyperexponential distribution is not.

communication demand in each iteration can be equally divided among (an arbitrary number of) available processors. These models address the performance impact of two aspects of such programs: (1) the scaling of communication and communication requirements with the number of processors, (represented by deterministic parametric functions), and (2) synchronization costs due to unequal (deterministic) waiting times incurred by the processors for communication resources (memory modules and the interconnection network). The models derive lower and upper bounds on speedup according to whether the processors need or need not synchronize at the end of each iteration.

The model of Tsuei and Vernon represents a program by a parallelism profile, where the phases of the profile are intervals of fixed parallelism derived from the program text, and the total computation requirement in each phase is obtained by measuring a sequential execution. (In practice, this restricts their model to fork-join programs.) In each phase the mean computational requirement as well as the mean overhead costs (communication, forking and lock contention) are assumed to be identical for all active processors. Thus, the model does not represent any synchronization costs in the program. The program execution time is estimated from the parallelism profile assuming processor-sharing in phases where the parallelism exceeds the number of processors used. This portion forms the high-level component of their model, while the simple queueing networks used to separately compute bus contention and lock contention together form the low-level model component. The model is shown to be accurate for three fork-join programs with good load balancing (i.e., in which ignoring synchronization delays at barriers does not introduce significant error).

We will not describe the other, less general, models for fork-join programs here, except to note that both the models of Heidelberger and Trivedi [HeT83] and Towsley et al [TRS90] apply to multiprogrammed parallel systems with multiple parallel jobs (each job is assumed to have the same number of tasks in each parallel phase and each task is exponentially distributed). All other models to which we refer in this paper only consider systems with a single executing job.

Finally, two previous models [LCB92, MaS91] are restricted to specific task graph structures. Madala and Sinclair [MaS91] propose a model that applies to divide-and-conquer task-graphs where tasks at each "level" in the graph have i.i.d. execution times with arbitrary variance, but with the assumption that the number of processors exceeds the maximum parallelism, i.e., task scheduling can be ignored. (They also derive models for fork-join programs that are very similar to the results of Kruskal and Weiss.) Lewandowski, Condon and Bach [LCB92] propose a model applicable to programs with pipelined task graphs and i.i.d. exponential task times.

2.5. Summary of the State of the Art

We can summarize what is known about the state of the art as follows. For fork-join programs, stochastic models with simplifying assumptions (particularly, i.i.d. task times and simplified task scheduling) have been developed that are efficient to solve [AIA90, DuB82, HeT83, KrW85, TRS90]. To our knowledge, these models have not been tested for accuracy using actual programs. Two efficient deterministic models [TsV90, VSS88] for restricted types of fork-join programs have also been developed, and shown to be accurate for several programs each.

Less restrictive analytical models, namely models that apply to non-fork-join programs and eliminate the above simplifying assumptions, have all assumed exponential task execution times for analytical tractability [KME89, MaL90, Moh84, ThB86]. Again, to our knowledge, none of these models has been tested for accuracy using actual programs, and validations against hypothetical task graphs have not tested the accuracy of the exponential task assumption. Furthermore, all these models use complex solution techniques, including state spaces that grow exponentially with the number of tasks in the models that are applicable to arbitrary task graphs, and no data is available showing the solution efficiency for actual programs.

2.6. Bounds on Parallel Program Performance

In addition to the models reviewed so far, techniques have been developed for computing bounds on the speedup of a program from a few key parameters describing the parallelism structure of the program. Amdahl's Law, based on the fraction of sequential work, is a well known example [Amd67]. Another important example is the set of bounds derived by Eager, Zahorjan and Lazowska, using primarily the *average parallelism* to characterize program parallelism [EZL89]. One of four equivalent definitions of the average parallelism of a program is the speedup of the program on an unlimited number of processors.[4] Given the average parallelism, A, they derived the

4. Note that in all references to "the program", we are specifically referring to the behavior of the program for a particular input set. Formally, this behavior is represented by a task graph. All their results require that the task graph be fixed, independent of the number of processors or the task scheduling algorithm used. Under these conditions, the average parallelism, defined above, is an intrinsic property of the task graph [EZL89].

following bounds which hold for any *work-conserving* task scheduling discipline:

$$\frac{PA}{P+A-1} \le \mathit{Speedup}(P) \le \min\{P,A\} \tag{2.1}$$

Furthermore, when these bounds hold, the geometric mean of the bounds lies within 34% of the true speedup, and thus A provides not only bounds but also an estimate of the speedup. They also showed that a tighter lower bound is possible for a specific task scheduling discipline, namely processor sharing, if the maximum parallelism, $P_{\max}$, is also considered:

$$\min\left\{A\ ,\ \frac{PA}{P+A-1-(P-1)(A-1)/(P_{\max}-1)}\right\} \le \mathit{Speedup}(P) \le \min\{P,A\} \tag{2.2}$$

(We will refer to these bounds as the $\langle A \rangle$ and $\langle A+P_{\max}|\mathrm{PS}\rangle$ bounds respectively.) Overhead costs such as due to communication could be represented by including them in the execution times of the tasks when computing A. It is important to note, however, that this is only possible for overhead costs that are fixed, independent of the number of processors. In particular, overhead due to shared resource contention cannot easily be included in this manner. Nevertheless, these bounds appear to provide a simple source of insight into program performance, and it is interesting to compare this insight with those available from a detailed analytical model. We do so in Chapter 6.

A second body of work also derives bounds on the execution time of a parallel program on a specific number of processors [HaM92, YaV91]. Unlike the bounds described above, these results are based on detailed inputs similar to those used in the stochastic models reviewed earlier, namely a full description of the task graph along with the distribution of the individual task execution times. Both approaches provide tight bounds for small graphs when the variance of task execution times is low, but the tightness of the bounds can decrease with the size of the input graph, and is also sensitive to the specific task graph structure. Furthermore, in contrast to previous models, both these techniques only apply to cases where the number of processors available is at least as large as the maximum parallelism in the input graph. Our discussion earlier in this Chapter shows that these techniques could still be applied to programs for which a condensed task graph can be derived (i.e., programs that use static scheduling of tasks). Within these limitations, however, these techniques hold the promise of providing useful bounds, especially when variance of task execution times is low. (Available data on the accuracy and efficiency is confined to small hypothetical task graphs.) In Chapter 5, we briefly discuss how these two bounding techniques compare with our more detailed modeling approach.

Chapter 3

The Influence of Random Delays on Execution Times in Parallel Programs

The execution of a parallel program on a multiprocessor system is influenced by a number of non-deterministic factors. In particular, inter-process communication events and contention for shared hardware and software resources introduce non-deterministic delays into the execution of a process in a parallel program. (These delays are non-deterministic not only in the sense that they are unpredictable, but also because different executions of the same program on the same input can experience different delays, even if the computation within the program is deterministic.) We refer to such delays as *random delays*. Consider an interval of execution of a process between consecutive synchronization points. Any random delays experienced by the process during such an interval makes the total length of the interval non-deterministic, potentially affecting the length of time processes must wait for each other at the subsequent synchronization point. Thus, random delays can affect program performance not only by increasing the mean length of the individual process execution times, but also by making these execution times variable, which potentially leads to higher synchronization costs during program execution.

In some programs, the CPU requirements can themselves be non-deterministic, i.e., they can vary significantly across different executions of the program for the same input. For example, in a program containing a heuristic search algorithm such as branch-and-bound, the specific sequence of computation in an execution can affect the *subsequent* computational work. Since the sequence of computation in a particular execution can itself be influenced by unpredictable communication and contention delays, such programs would have non-deterministic CPU requirements.[5]

Performance models of parallel programs and systems have typically used stochastic task or process execution times to represent non-determinism due to these various sources, as the

5. Non-deterministic computational behavior can also arise because of algorithms that are intrinsically random, e.g., based on a true random-number generator. We do not include this specialized class when we refer to "programs with non-deterministic CPU requirements."

discussion in Chapters 1 and 2 indicates. In many parallel programs, however, the CPU requirements *are* fixed or almost fixed for any particular input. (Some evidence for this is provided later in the chapter.) For such programs, does the variability due to random delays justify the stochastic assumptions in these performance models? More generally, how do random delays influence the variance of execution time of synchronizing processes in parallel programs?

The answers to these questions should be useful not only for performance evaluation, but for programmers as well. For example, in current systems, static scheduling of tasks is often considered adequate when the programmer believes that the processing requirements can be evenly divided among the available processors. However, if communication and contention delays introduce significant variance into the execution time of a process between synchronization points, dynamic scheduling of tasks may be necessary to provide adequately balanced execution times across the processes. Thus, an understanding of the magnitude of variability in execution times due to random delays could be directly useful during parallel program development as well. To our knowledge, however, the above questions have not previously been addressed.

In this chapter, we use an analytical model of program behavior parameterized with detailed program measurements to address these questions, and we briefly explore the implications of the results for performance evaluation as well as for parallel programming. We first describe a renewal model of program behavior that can be used to evaluate the variance and distribution of the execution time of a process between synchronization points, in the presence of random delays (Section 3.2). The model yields a simple estimate for the variance in terms of basic and intuitive parameters. In Section 3.3, we apply the model to different phases of several shared-memory programs, using detailed measurements to obtain the necessary parameter values. We also present direct measurements of the variance and distribution of process execution times, which include variability due to processing requirements as well. We use these to compare the relative influence of the two sources of non-determinism, as well as to evaluate the overall variance and distribution of execution time.

In subsequent chapters, we discuss what the results of this study imply for parallel program performance prediction models [KME89, KrW85, MaL90, Nel90, ThB86] as well as for more general stochastic models of parallel systems [BaL90, ChN91, LeV90, LeN91, LCB92, NTT88, Nel90, NTT90, SeT91, ZaM90]. In the former case, the results motivate an approach to analytical parallel program performance prediction that is different from most previous models for this purpose. Developing, validating and demonstrating the usefulness of this approach form the principal subject of subsequent chapters in this thesis. In the latter case, our work implies that it could be particularly important to evaluate the effect of stochastic model assumptions on model results. In particular, our work shows that simplifying assumptions such as that of exponential task or process execution times do not reflect real program behavior in many programs. Thus, it is important to determine if a result of a parallel system performance model is strongly dependent on the exponential task assumption, since such a result will not be applicable to many parallel programs. We discuss this in a little more detail in Chapter 8, giving one example each of previous results that are and are not strongly dependent on such an assumption. In that chapter, we also briefly discuss an implication of our results for programmers of parallel systems.

3.1. Preliminaries and Related Work

We begin with a discussion of some key terms and concepts, and a review of the few related results and comments by previous authors. The terms *task*, *process* and *synchronization point* were defined in Table 2.1; all three definitions are key to the results in this chapter. A process is the logical entity that executes the tasks of a program. A process will occasionally be required to synchronize with one or more other processes to enforce the precedences between tasks; such a point during the execution of a process is called a *synchronization point*. In particular, this excludes accesses to synchronization objects such as locks or monitors that are made solely for the purpose of mutual exclusion. (Note that such accesses do not introduce precedences in the task graph.) The delays due to such accesses will be represented as random delays due to shared resource contention, rather than as synchronization points.

We emphasize that the focus of this work is the behavior of a parallel program (in particular, the variance and distribution of execution times) *for a single input data set*, rather than across different input sets. This is consistent with our goal of understanding the influence of non-determinism on synchronization costs within a parallel program.

To our knowledge, there has been no previous attempt to study the effect of random delays on the variance or distribution of execution times. However, one previous paper focuses on estimating the mean and variance of the processing requirements of tasks in the presence of data-dependent effects such as conditional branch probabilities and loop frequencies [Sar89]. In that work, Sarkar describes a framework for determining the mean and variance of task execution times using frequency information from a counter-based execution profile of the program. For example, his method could be applied to a particular loop to estimate the mean and variance of the execution time of successive iterations of the loop, due to the above data-dependent effects. In contrast, the goal of our study is to evaluate the variability in total execution time of a particular iteration or set of iterations due to the various sources of non-determinism described above.

Finally, stochastic models that allow general distributions of task-time have been applied using different specific distributions, including the normal distribution. [DuB82, Gre89, KrW85]. Dubois and Briggs [DuB82] as well as Greenberg [Gre89] argued that a task could be asymptotically normally distributed because it is the sum of a large number of (non-deterministic) instruction execution times. Our proof in Appendix A is essentially a formalization of this argument.

3.2. Renewal Model of the Effect of Random Delays

In this Section, we provide a framework for analyzing the variance and distribution of execution time attributable to random delays. We first describe a model of process behavior, and the assumptions in our analysis. In Section 3.2.1, we derive exact and approximate expressions for the variance, and use the exact expression to validate the approximation. We also use the approximate expression to study the variance for hypothetical values of the model parameters. In Section 3.2.2, we use the same model to derive the asymptotic distribution of execution time.

We consider a program executing on a parallel system, and focus on an interval in the execution of one process. In Section 3.3, we will apply the model to intervals between synchronization points, but the model and analysis in this section apply to other intervals such as a single task execution as well. Let D denote the total CPU requirement of the process in this interval, and let the random variable T denote the length of the interval, i.e., the total time to complete this processing requirement. In general, the execution of the process in the interval consists of a sequence of alternating processing and delay sub-intervals, where each delay represents a sub-interval in which the process busy-waits or is suspended (for example, for remote communication, access to a critical section, or other accesses to shared resources). Denoting the number of intervals required to complete the total processing requirement (D) by R, the lengths of successive processing sub-intervals by $\{P_i, i \geq 1\}$ and the lengths of delay sub-intervals by $\{C_i, i \geq 1\}$ with $C_\Sigma \equiv \sum_{i=1}^{R} C_i$, the total interval length is given by:

$$T = P_1 + C_1 + P_2 + C_2 + \cdots + P_R + C_R + P_{R+1} \qquad (1)$$

We assume (1) that the $2{\times}R$ random variables $\{P_i : 1 \le i \le R\}$ and $\{C_i : 1 \le i \le R\}$ are mutually independent, (2) $\{P_i : 1 \le i \le R\}$ have common distribution F_P, (3) $\{C_i : 1 \le i \le R\}$ have common distribution F_C, and (4) that each of these distributions has finite variance. In practice, in the presence of resource contention and non-stationary behavior (such as a burst of cache misses as each new task begins execution) the first three assumptions may be violated. One goal of the measurements in Section 3.3 is to validate the accuracy of the model results. Without these assumptions, the process behavior becomes much more difficult to analyze, and the solution would require more complex (and less intuitive) parameter values that would be difficult to measure for real programs.

For our further analysis of the model, we assume that D has zero variance, i.e., that D is constant for this program interval. This assumption is made because the goal of this analysis is to study the variability due to random delays alone. (The implications of this assumption are discussed below.) The random variable R and the sequence of processing times $\{P_i : 1 \le i \le R{+}1\}$ must satisfy $\sum_{i=1}^{R+1} P_i = D$. Under these assumptions, denote the distribution, mean, variance and coefficient of variation of the total execution time T by $F_{T|D}$, $\mu_{T|D}$, $\sigma^2_{T|D}$ and $\mathrm{CV}_{T|D}$, respectively.[6] Since D is assumed to be constant, $\mathrm{CV}_{T|D} \equiv \sigma_{T|D}/\mu_{T|D} = \dfrac{\sigma_{C_\Sigma|D}}{(D + \mu_{C_\Sigma|D})}$. Thus, $\mathrm{CV}_{T|D}$ is a measure of (normalized) variability in the execution time of the interval due to random delays. This is the principal measure we will use to understand the influence of random delays on specific programs, when we apply the model to these programs in Section 3.3.

If the assumption that D is constant is true for a particular program interval, then $\mathrm{CV}_{T|D}$ is the coefficient of variation of execution time of that program interval. If the value of D can vary across different executions (for example, if the instructions executed in a task depend on which tasks have previously been completed by other processes), D itself can be influenced by the random delays experienced by this process as well as other processes of the program. However, if the variability of D is small relative to the average length of the interval, $\mathrm{CV}_{T|D}$ can still be used to give a qualitative estimate of the impact of random delays. Thus, for such programs, it will be important to understand to what extent the value of D for this interval can vary across different executions of the

6. In general, we use μ_X, σ^2_X, CV_X and F_X to denote the mean, variance, coefficient of variation and distribution function of the random variable X ($\mathrm{CV}_X \equiv \sigma_X/\mu_X$). We denote the Laplace transform of (the distribution of) X as $\mathcal{F}_X$.

program.

3.2.1. Analysis of the Normalized Variance $\mathrm{CV}_{T|D}$

The main goal of this section is to derive an expression for $\mathrm{CV}_{T|D}$ in terms of five input parameters: D, μ_P, σ_P^2, μ_C and σ_C^2, where the latter four terms denote the mean and variance of $\{P_i : 1 \le i \le R\}$ and the mean and variance of $\{C_i : 1 \le i \le R\}$ respectively. We first derive $\mu_{T|D}$ and $\sigma_{T|D}^2$ in terms of μ_R and σ_R^2, and then derive μ_R and σ_R^2 in terms of the other input parameters. Note that R is determined only by $\{P_i\}$, i.e., it is independent of $\{C_i\}$. Hence, for $x \ge D$, we can write the distribution of T, $F_{T|D}(x) \equiv \mathrm{P}\{T \le x\}$, as

$$F_{T|D}(x) = \mathrm{P}\{R{=}0\} + \sum_{k=1}^{\infty} \mathrm{P}\{R{=}k\}\mathrm{P}\,\{\textstyle\sum_{i=1}^{i=k} C_i \le x - D\},\ \ x \ge D$$

$$= \mathrm{P}\{R{=}0\} + \sum_{k=1}^{\infty} \mathrm{P}\{R{=}k\} F_C{}^{(k*)}(x{-}D),\ \ x \ge D$$

where $F_C{}^{(k*)}$ denotes the k-fold convolution of F_C with itself. (Note $F_{T|D}(x){=}0$, $x{<}D$.) On taking transforms, we obtain:

$$\mathcal{F}_{T|D}(s) = \sum_{k=0}^{\infty} \mathrm{P}\{R{=}k\} e^{-Ds}\, \mathcal{F}_C{}^k(s)\ . \tag{2}$$

Differentiating (2) and using $\mathrm{E}[T^k] = (-1)^k \frac{d^k}{ds^k} \mathcal{F}_{T|D}(s)\big|_{s=0}$ gives $\mu_{T|D}$ and $\sigma_{T|D}^2$ in terms of μ_R and σ_R^2:

$$\mu_{T|D} = D + \mu_R\, \mu_C \tag{3a}$$

$$\sigma_{T|D}^2 = \mu_R\, \sigma_C^2 + \mu_C{}^2\, \sigma_R^2 \tag{3b}$$

Although (3) gives exact expressions for $\mu_{T|D}$ and $\sigma_{T|D}^2$, it would be impractical to directly apply these to a particular interval of execution of a real program because of the difficulty of measuring μ_R and σ_R^2 directly. Specifically, this would require a large number of measured samples of R for that interval, i.e., it would require repeated detailed measurements of the interval in a number of executions of the program. Instead, our approach is to obtain analytical estimates of μ_R and σ_R^2 in terms of D, μ_P and σ_P^2, and thus use (3) to obtain expressions for $\mu_{T|D}$ and $\sigma_{T|D}^2$ in terms of D, μ_P, σ_P^2, μ_C and σ_C^2. These five parameters can be estimated by measuring the relevant interval in a *single* execution of the program, and repeated detailed measurements would not be necessary. We therefore focus on obtaining exact and approximate expressions for μ_R and σ_R^2.

3.2.1.1. Exact derivation for $\mu_{T|D}$ and $\sigma_{T|D}^2$

The analysis of R is complicated by the dependence between $\{P_i : 1 \le i \le R\}$ and P_{R+1}. We can avoid this difficulty by re-casting R as a function of a sequence of independent, identically distributed RVs $\{P_i' : 1 \le i < \infty\}$, each having the same distribution as each of $\{P_i : 1 \le i \le R\}$:

$$R(t) \equiv \max \{r \ge 0 : \textstyle\sum_{i=1}^{i=r} P_i' \le t\} \tag{4}$$

Then R, as defined earlier, has the same distribution as $R(D)$, and $P_{R+1} = D - \sum_{i=1}^{R(D)} P_i$. But (4) is just the definition of a *renewal process* generated by $\{P_i' : 1 \le i < \infty\}$; specifically, $R(t)$ is the number of renewals by time t [Wol89].

The following expressions for $\mu_R(t) \equiv \mathrm{E}[R(t)]$ and its ordinary Laplace transform are well-known [Wol89]:

$$\mu_R(t) = F_P(t) + \mu_R(t) * F_P(t),$$

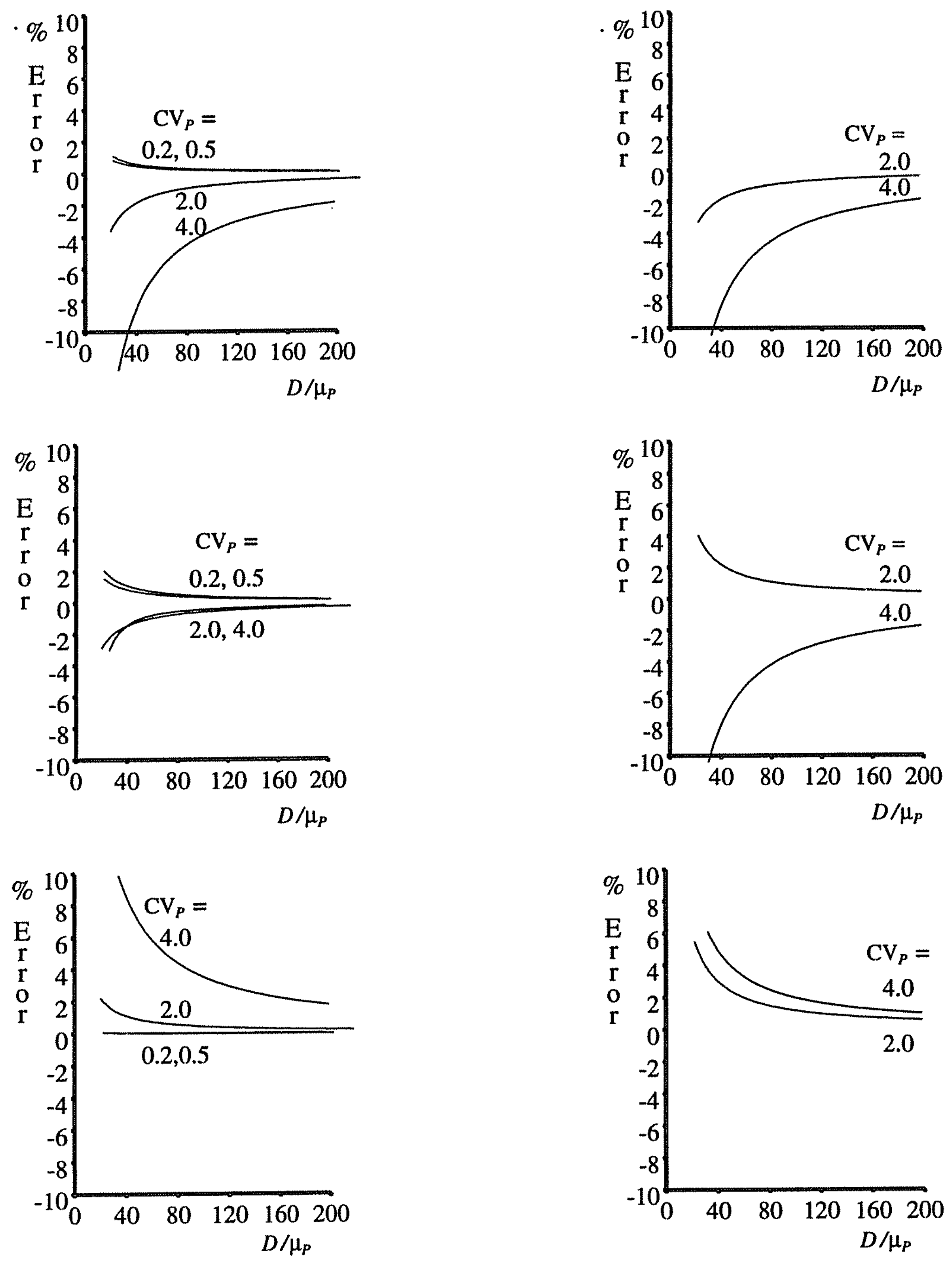
$$\mathcal{L}(\mu_R(t)) = \frac{\mathcal{F}_P(s) / s}{1 - \mathcal{F}_P(s)}, \tag{5}$$

The corresponding expressions for $\mu_{R^2}(t) \equiv \mathrm{E}[R^2(t)]$ and its Laplace transform are also not difficult to derive [Wol89]:

$$\mu_{R^2}(t) = \mu_R(t) + 2\int_0^t \mu_R(t-x)\, d\mu_R(x),\ t \ge 0$$

$$\mathcal{L}(\mu_{R^2}(t)) = \frac{(1 + \mathcal{F}_P(s))\ \mathcal{F}_P(s) / s}{(1 - \mathcal{F}_P(s))^2} \tag{6}$$

where $\mathcal{L}(\mu_R(t))$ and $\mathcal{L}(\mu_{R^2}(t))$ denote the ordinary Laplace transforms of $\mu_R(t)$ and $\mu_{R^2}(t)$ respectively, and $*$ denotes the convolution operator. Equations (5) and (6) together allow us to compute $\mu_R(D)$ and $\sigma_R^2(D)$ for a particular distribution $F_P(t)$, but this generally requires numerical inversion of the Laplace transforms. A more practical application of (5) and (6) is to validate simpler approximations for μ_R and σ_R^2, which we derive next.



(a) F_P: Gamma Distribution (b) F_P: Hyper-Exponential Distribution ($p_1 = 0.9$)

Figure 3.1. Relative error in the approximations for Mean $\mu_{T|D}$ (top), Variance $\sigma^2_{T|D}$ (center), and Coefficient of Variation $CV_{T|D}$ (bottom)

$\mu_C = \mu_P = 0.5$, $CV_C = 2.0$

3.2.1.2. Approximate expressions for $\mu_{T|D}$ and $\sigma^2_{T|D}$

Estimates for $\mu_R(t)$ and $\sigma^2_R(t)$ are given in the Central Limit Theorem for renewal processes [Wol89], which states that as $t \to \infty$, $R(t)$ is asymptotically normal with mean[7] t/μ_P and variance $t\sigma^2_P / \mu_P{}^3$. Therefore, we estimate σ^2_R by $D\sigma^2_P / \mu_P{}^3 = (D/\mu_P)\mathrm{CV}^2_P$. Using these estimates for μ_R and σ^2_R in (3) gives us our final approximate expressions for the mean and variance of T:

$$\mu_{T|D} \approx D\,(1 + \frac{\mu_C}{\mu_P}), \quad \sigma^2_{T|D} \approx \frac{D}{\mu_P}\,\mu_C{}^2\,(\mathrm{CV}^2_C + \mathrm{CV}^2_P), \tag{7a}$$

i.e,

$$\mathrm{CV}_{T|D} \approx \frac{1}{\sqrt{D/\mu_P}}\,\frac{\mu_C}{\mu_C + \mu_P}\sqrt{\mathrm{CV}^2_C + \mathrm{CV}^2_P} \tag{7b}$$

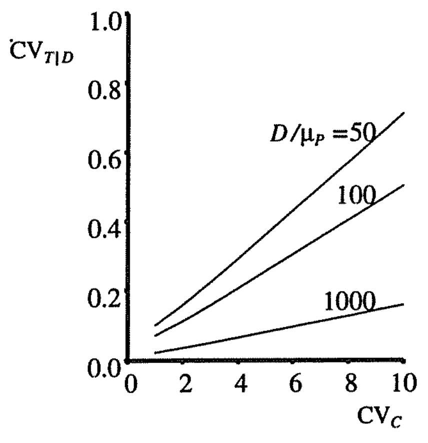
The key terms in this expression for $\mathrm{CV}_{T|D}$ include the average number of sub-intervals: $D/\mu_P \approx \mu_R$, the average fraction of time the process is delayed: $\mu_C/(\mu_C+\mu_P)$, and a term representing the variability of the individual processing and delay sub-intervals.

The above expressions for $\mu_{T|D}$, $\sigma^2_{T|D}$ and $\mathrm{CV}_{T|D}$ are only asymptotically exact, but they can be compared at finite D against exact values calculated using (5) and (6), for specific distributions F_P, and specific values of μ_P, μ_C, CV_P and CV_C. We do so in Figure 3.1 using gamma and 2-stage hyperexponential distributions for F_P, for $\mu_C/\mu_P = 1$, $\mathrm{CV}_C = 2.0$ and for a range of values of CV_P. The figure shows that for $\mu_C/\mu_P = 1$, the error in the approximation is potentially significant when $D/\mu_P < 40$ and CV_P or $\mathrm{CV}_C > 4$, but is likely to be acceptably low otherwise. The error increases when μ_C/μ_P increases (not shown), and $\mu_C = \mu_P$ is conservative relative to measured parameter values in Section 3.3.

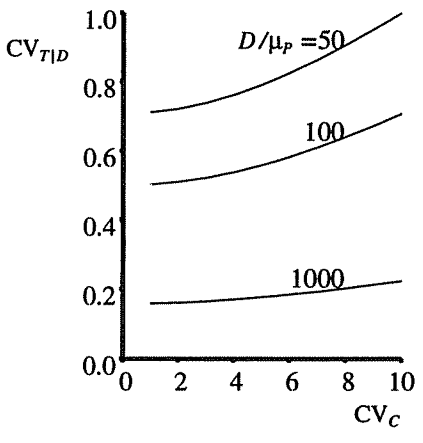
3.2.1.3. Quantifying the Variance due to Random Delays

A key observation from (7) is that $\mathrm{CV}_{T|D}$ decreases as $1/\sqrt{\mu_R} = 1/\sqrt{D/\mu_P}$. Thus, for intervals containing a very large number of delays, we expect the total delay time to have very little overall variability relative to T. (Intuitively, the individual fluctuations of a large number of delay times will tend to cancel each other out in the long run.) We can use (7) to quantify how large D/μ_P must be for this observation to hold, for various values of the other parameters. In Figure 3.2, we again set $\mu_C = \mu_P$ and show $\mathrm{CV}_{T|D}$ for a wide range of CV_P and CV_C. (Note that CV_P and CV_C are

7. In fact, this expression for the mean is exact for all $t > 0$ if, and only if, the first interval P_1 has distribution equal to the *distribution of residual life* of $P_i, i \geq 2$. For example, this holds automatically for the exponential distribution.



(a) $CV_P = 1$



(b) $CV_P = 10$

Figure 3.2. Effect of random delays on relative variability of total delay ($CV_{T|D}$).

$\mu_P = \mu_C = 1$

interchangeable as far as their influence on $CV_{T|D}$ is concerned.) The graphs show that $CV_{T|D}$ can be high (>0.5) when CV_P or CV_C is very high (≥10) and the interval contains 100 or fewer delays. For intervals with 1000 or more delays, however, $CV_{T|D}$ is low even when CV_P and CV_C are as large as 10 and $\mu_C = \mu_P$. Furthermore, when CV_P and CV_C are close to 1, even intervals containing as few as 50 delays have very low $CV_{T|D}$.

The above arguments are inconclusive about whether random delays cause significant variability in actual programs, since it is unknown what values of the model parameters occur in practice. One goal of the measurements presented in Section 3.3 is to obtain these parameter values for real programs, and show where in the parameter space typical programs can be expected to lie.

3.2.2. Asymptotic Analysis of the Distribution $F_{T|D}$

We were able to obtain explicit expressions for $\mu_{T|D}$ and $\sigma^2_{T|D}$ in terms of the Laplace transforms of μ_R and σ^2_R. Obtaining general expressions for the *distribution* of T is difficult. We can, however, derive the form of the distribution in the special case when D is large. In that case, we show in Appendix A that it is possible to apply the version of the Central Limit Theorem for cumulative (regenerative) processes [Wol89] to prove:

$$T(D) \Longrightarrow \text{Normal}(\mu(D), \sigma(D)) \text{ as } D \rightarrow \infty, \text{where}$$

$$\mu(D) = D + \frac{D\,\mu_C}{\mu_P}, \tag{8}$$

$$\sigma^2(D) = \frac{D\sigma_C^2}{\mu_P} + \frac{{\mu_C}^2 D\sigma_P^2}{{\mu_P}^3},$$

and $\Longrightarrow$ denotes convergence in distribution.

Two points are worth noting here. First, the mean, $\mu(D)$, and variance, $\sigma^2(D)$ *are the same as* $\mu_{T|D}$ *and* $\text{CV}_{T|D}$ *calculated in (7)* using the estimate for μ_R and σ_R^2. Second, there is an important difference between the estimates for μ_R and σ_R^2 and the asymptotic normal distribution of T calculated in (8): the convergence in (8) depends on σ_C^2, whereas the estimates of μ_R and σ_R^2 did not. Thus, high variance of communication delays could make a moderately large task look different from normal, but the estimate for μ_R and σ_R^2, and hence the estimate in (7) for $\mu_{T|D}$ and $\sigma_{T|D}^2$, could still be accurate.

3.3. Applications of the Model

In this section, we address the questions raised in Section 3.2, using data obtained from measurements of parallel programs. We begin with a description of the applications and measurement methodology in Section 3.3.1. In Section 3.3.2, we apply the renewal model to study the impact of random delays on these programs, by using the data to show where in the parameter space of the model these programs lie. In Section 3.3.3, we examine whether the assumptions of the model introduce significant errors into the qualitative conclusions obtained. While addressing this question, we measure the overall variance in process execution times, due to both random delays and variation in processing requirements. This also allows us to comment on the total variance found in practice. Finally, in Section 3.3.4 we examine whether real programs exhibit normally distributed process execution times in practice.

3.3.1. Applications and Measurement Methodology

We measured a variety of applications on a 20-processor Sequent Symmetry S-81, as well as a few shared-memory applications running on the Thinking Machines CM-5. Table 3.1 gives a brief overview of the applications and inputs. (*Small* and *Large* are mainly labels we will use for

Table 3.1. Applications used for the Measurement Experiments.

Name	Application	Phase Structure	Dominant Phases	Input Data
MP3D	Hypersonic flow simulation	Five phases with intervening barriers	*Move* : more than 90% of total work	*Small*: 5000 mols. *Large*: 20000 mols.
Locus Route	Standard cell wire routing	Two iterations, parallel loop per iteration	-	*Small*: bnrE
Water	Water molecule simulation	One large, several small phases; intervening barriers (per iteration)	*Inter-molecular force evaluation*: almost all the work.	*Small*: 64 mols. *Large*: 343 mols.
Barnes	Gravitational N-body simulation	One large, several small phases	*Force computation*: 90% of total work	*Small*: 1024 bodies *Large*: 8192 bodies
Bicon	Graph Biconnectivity	More than 50 phases; intervening barriers	-	*Large*: 4096 nodes, 16384 edges
Hydro	Particle motion in viscous fluids	N parallel loops (N = no. of particles)	-	*Toy*: 2 particles *Small*: 8 particles
PSIM	Multistage network simulation	Single parallel phase per iteration	-	*Small*: 1024 nodes *Large*: 4096 nodes

convenience. The *Large* input size is a somewhat more realistic data set than the *Small* size.) Four of the applications (MP3D, Locus Route, Water and Barnes) are from the Splash suite, which was developed to provide a realistic set of parallel applications for performance evaluation of parallel systems [SWG92]. The other three are also real applications in the sense that they were written to solve computationally intensive problems of interest to their authors. Hydro is a parallel simulation of particle motion in viscous fluids, with efficient communication. [FuK92]. PSIM was developed at Lawrence Livermore Laboratories to simulate the indirect binary n-cube memory server network in a large parallel vector-processing environment [Bro88b]. Bicon is an implementation of a parallel algorithm to find the biconnected components of large graphs [TaV85].

We measured each of the programs running stand-alone, allowing us to characterize the nondeterminism intrinsic in the program. This is important because parallel system models (such as those for parallel program performance prediction or for analysis of scheduling policies) require parameters that characterize the intrinsic behavior of the program as input. It is also worth noting that intrinsic random delays are the key unknown for determining synchronization costs in multiprogrammed systems where the processes of an application are (essentially always) co-scheduled. The intrinsic random delays in the applications of Table 3.1 are communication delays due to remote memory accesses. In particular, page faults and lock contention did not cause significant delays in these programs.

For each application, we focused on one to three phases of the program, where a phase is bounded by barrier synchronizations, and has no intervening synchronization points. We measured one process in each phase, and the interval length T corresponds to the execution time of the measured process between the corresponding barriers. D is the total processing requirement in that interval, and the delay sub-intervals $\{c_i\}$ are the remote communication delays. The experiments in Sections 3.3 and 3.4 require repeated measurements of a particular phase; exactly the same input data set is used for each such measurement, as explained in Section 3.1.

3.3.2. Measurements for Evaluating Execution Time Variance

Before presenting the results for the evaluation of variance due to random delays, it is important to recall that $\mathrm{CV}_{T|D}$ (the measure of interest) will be estimated for a program phase from a single execution of the program, as explained below equation (3). As discussed in Section 3.2, for program phases in which the processing requirement (D) does not vary across different runs, $\mathrm{CV}^2_{T|D}$ is the normalized variance in execution time of the interval, due to random delays. For phases in which D varies across runs but the variability in D is small relative to the average length of the interval (see Section 3.3.3), $\mathrm{CV}_{T|D}$ should still give a qualitative estimate of the influence of random delays on the variance of execution time.

3.3.2.1. Measurements on the Sequent Symmetry

Shared-memory on the Sequent Symmetry is supported by an invalidation-based snooping cache protocol, and the communication delays are due to three types of remote requests (*read-shared*, *read-invalidate* and *invalidate*). The measurements were made using non-intrusive hardware probes to record the relevant bus events.[8] The various parameter values were subsequently derived from the stored traces.

8. A Tektronix DAS 9200 Logic Analyzer was used for this purpose.

Table 3.2. Measurements of Renewal Model Parameters: Sequent Symmetry†.

Application			Measured Parameter Values								Model Results	
Program		Phase	Input	D ($\times 10^3$)	μ_P	μ_C	CV_P	CV_C	$R = \frac{D}{\mu_P}$	$\frac{\mu_C}{\mu_P+\mu_C}$	$\mu_{T\|D}$ ($\times 10^3$)
MP3D	*Res-Move*	Small	7.7	236.9	12.00	2.00	1.68	33	0.05	8.0	0.0221
	Res-Coll	Small	11.7	182.0	9.90	1.38	1.30	64	0.05	12.4	0.0122
	Move	Small	608.2	410.1	14.6	1.03	0.80	1483	0.03	629.8	0.0012
	Res-Move	Large	26.3	316.5	11.60	1.34	1.49	83	0.04	27.2	0.0078
	Res-Coll	Large	33.7	267.8	11.30	1.13	1.26	126	0.04	35.1	0.0061
	Move	Large	2396.7	349.0	12.9	1.06	0.43	6867	0.04	2485.1	0.0005
PSIM	-	Small	325.5	76.2	11.1	0.86	0.50	4272	0.13	372.8	0.0019
	-	Large	1507.9	66.8	10.8	0.74	0.44	22573	0.14	1752.2	0.0008
Bicon	*Connl*	Large	191.7	147.5	23.5	2.23	0.79	1300	0.14	222.3	0.0090
	Lowhigh	Large	279.9	217.4	8.7	1.15	0.51	1287	0.04	291.1	0.0014
	Tour	Large	3785.2	72.7	23.4	4.82	0.67	52066	0.24	5003.54	0.0053
Locus	*Itn. 2*	Small	2159.8	231.4	10.60	3.58	0.53	9334	0.04	2258.9	0.0016
Route	*Itn. 1*	Small	2904.1	137.9	14.70	4.02	0.80	21059	0.10	3214.1	0.0027
Water	*Inter-mol*	Small	4203.0	1584.8	9.9	3.71	0.36	2652	0.006	4229.1	0.0004
	Inter-mol	Large	122667.6	3767.5	10.50	2.55	0.34	32559	0.003	123009.4	0.0000
Barnes	*Force*	Small	16577.9	4781.6	8.0	2.42	0.34	3467	0.002	16605.6	0.0001
	Force	Large	223443.5	3073.2	7.6	2.22	0.26	72707	0.002	223995.2	0.0000
Hydro	*Nucleus*	Toy	102967.3	9662.8	8.5	4.35	0.36	10656	0.001	103057.8	0.0000
	Nucleus	Small	336718.3	6735.9	8.0	2.73	0.33	49989	0.001	337120.0	0.0000

† μ_P, μ_C and R are in units of bus cycles (0.1 microseconds).
D and $\mu_{T|D}$ are in units of 1000 bus cycles (100 microseconds).

Table 3.2 gives the measured parameter values, as well as the model results ($\mu_{T|D}$ and $CV_{T|D}$), for each of the application phases executing on 16 processors on the Sequent. Values for μ_P and μ_C are given in units of bus cycles (equal to 0.1 microseconds), and for D and $\mu_{T|D}$ in units of 1000 bus cycles (100 microseconds). The applications are listed in generally increasing order of processing demand D, although the demand varies from phase to phase in each application. In addition to the five basic input parameters, we also give the values of $R = D/\mu_P$ (the measured number of random delays) and $\mu_C/(\mu_C+\mu_P)$.

Before interpreting the results obtained, two points are worth noting about the measured parameters in Table 3.2. First, the values fall in the region of the parameter space where the approximate solution of the renewal model is expected to be accurate (Figure 3.1). Second, the measured application phases vary widely in terms of all five input parameters, as well as the number of random delays (D/μ_P) and communication overhead ($\mu_C/(\mu_C+\mu_P)$). In particular, the total processing demand (D) varies by more than 4 orders of magnitude (from 770 microseconds for the *Res−Move* phase of MP3D to more than 33 seconds for Hydro), the number of random delays varies (somewhat in proportion to D) from 33 to 72707, CV_P and CV_C are as high as 4.35 and 1.68 respectively, and communication overhead is as high as 0.24 (Bicon). Thus, the measured applications have widely varying behavior, and communication requests experience significant variability.

The most striking observation from the table is that in every one of these application phases, the predicted variability of execution time due to random delays ($CV_{T|D}$) is extremely low. The highest estimated $CV_{T|D}$ on this system is 0.022 (for the *Res-Move* phase of MP3D) and this was obtained with a much smaller input size than expected in practice [SWG92]. To analyze these results further, we compare the measured parameters against regions of the parameter space in Figure 3.2 (recalling that Figure 3.2 is pessimistic for communication overhead less than or equal to 0.24). In all but a few of the cases $D/\mu_P > 1000$ and, as shown in Figure 3.2, this alone explains why the predicted $CV_{T|D}$ is extremely low for those cases. Thus, D/μ_P has a dominant role in determining $CV_{T|D}$ in most cases on this system. In the few cases where D/μ_P is on the order of 100 or less (and in many other cases in the Table), CV_P and CV_C are both less than 2. Thus, again, $CV_{T|D}$ is extremely low. In fact, for the worst case combination of parameter values across all the measured application phases ($D/\mu_P = 33$, $\mu_C/(\mu_C+\mu_P) = 0.24$, $CV_P = 4.82$ and $CV_C = 1.68$), the renewal model predicts that $CV_{T|D}$ would be 0.21, which is still low.

We next consider how these parameter values could change for applications on highly parallel shared-memory systems. First, the range of D/μ_P seen here (more than 4 orders of magnitude) could be representative of larger systems as well. Large values of D/μ_P are still likely to occur due to scaling up problem sizes, yet the fraction of applications that have small D/μ_P could also increase if finer granularity of synchronization is supported efficiently in future systems, or if applications can use new primitives to completely overlap a significant fraction of communication events with computation. In applications for which D/μ_P is as large as in most cases seen here, $CV_{T|D}$ should continue to be low, because the effect of $\mu_C/(\mu_C+\mu_P)$, CV_P and CV_C would have to be one to two orders of magnitude higher than observed here to yield even $CV_{T|D} \geq 0.1$. In applications where D/μ_P is significantly lower, increases in $\mu_C/(\mu_C+\mu_P)$, CV_P or CV_C are important to consider. It is not clear how these parameters would extrapolate to other applications and systems. Efficiency considerations alone dictate that $\mu_C/(\mu_C+\mu_P)$ will generally be less than about 0.5 (i.e., 50% efficiency).[9] CV_P arises due to the non-uniform intervals between cache misses and there do not appear to be reasons to believe that this will be significantly different in future systems. We next discuss two further experiments that were designed with the goal of producing higher values of CV_C and $\mu_C/(\mu_C+\mu_P)$, which could exist in future parallel systems. In any case, the model demonstrates the need to obtain these parameter values, and provides a framework for estimating the influence of these parameters as future systems become available.

3.3.2.2. Measurements on the Sequent with External Bus Load

For this experiment, we wrote an artificial parallel program to increase the bus contention. All tasks of the "bus-loading" program repeatedly read and write a fixed memory location causing that cache line to bounce from cache to cache. We repeated the measurements with MP3D, PSIM or Bicon running on 4 processors, and the bus loading program on 14 other processors. (We used the smaller input size in each case, which gives approximately the same total work per process per phase as the larger input on 16 processors). The results for these measurements are given in Table 3.3. Comparing with the numbers in Table 3.2, we see that $\mu_C/(\mu_C+\mu_P)$ is much higher, yet CV_C is not significantly different despite the increased contention. Thus, the highest estimated value of

9. On multithreaded processors, however, $\mu_C/(\mu_C+\mu_P)$ could be somewhat higher and still allow reasonable efficiencies.

Table 3.3. Measurements of Renewal Model Parameters: Sequent Symmetry with Bus Load[†].

Application			Measured Parameter Values							Model Results	
Program		Phase	Input	D ($\times 10^3$)	μ_P	μ_C	CV_P	CV_C	$R = \frac{D}{\mu_P}$	$\frac{\mu_C}{\mu_P+\mu_C}$	$\mu_{T\mid D}$ ($\times 10^3$)
MP3D	*Res-Move*	Small	26.3	315.7	45.80	1.66	0.75	83	0.13	30.1	0.0253
	Move	Small	2342.0	408.7	59.40	0.99	0.67	5730	0.13	2682.6	0.0020
PSIM	-	Small	1303.9	71.9	46.1	0.76	0.82	18134	0.39	2139.8	0.0032
Bicon	*Connl*	Small	162.8	89.7	49.80	1.84	0.70	1815	0.36	253.2	0.0165
	Tour	Small	6443.5	81.1	54.1	0.94	0.68	79451	0.40	10741.8	0.0016

† μ_P, μ_C and R are in units of bus cycles (0.1 microseconds).
D and $\mu_{T\mid D}$ are in units of 1000 bus cycles (100 microseconds).

Table 3.4. Measurements of Renewal Model Parameters: CM-5 + Wisconsin Windtunnel[†].

Application			Measured Parameter Values							Model Results	
Program		Phase	Input	D (ms)	μ_P (μs)	μ_C (μs)	CV_P	CV_C	$R = \frac{D}{\mu_P}$	$\frac{\mu_C}{\mu_P+\mu_C}$	$\mu_{T\mid D}$ (ms)
MP3D	*Move*	Small	44.82	49.3	1551.5	6.18	0.88	909	0.969	1454.86	0.2006
	Move	Large	147.84	42.0	1508.5	3.68	0.76	3520	0.973	5451.10	0.0617
Water	*Inter-mol*	Small	253.8	564.0	794.8	4.12	1.75	450	0.585	611.5	0.1234
	Inter-mol	Large(512)	6729.27	376.8	771.9	3.26	0.50	17859	0.672	20514.63	0.0166

† μ_P, μ_C and R are in units of 1 microsecond. D and $\mu_{T\mid D}$ are in units of 1000 microseconds.

$CV_{T|D}$ seen here is still only 0.025.

3.3.2.3. Measurements on the CM-5

We were also able to measure two of the above programs on a 32-processor Thinking Machines CM-5, a system that is scalable to much larger numbers of processors. Since the CM-5 does not support shared memory, we used the Wisconsin Windtunnel [RHL93] to simulate the execution of the shared-memory applications. In this simulator, the application program executes most of the time at full hardware speed on the processing nodes of the CM-5, but traps into software on memory references to cache blocks that would not be in the target machine's cache. Explicit messages are used to obtain remote cache blocks. As on the Sequent, the delays for remote communication were the delay intervals, $\{C_i : 1 \le i \le R\}$. Since every remote communication event causes a trap into the software handler on the local node, the measurements are done in software. In measuring the processing intervals, we ensured that the time to service remote memory requests (i.e. interrupts) from other nodes was excluded. Other than these interrupts the application runs at full hardware speed between traps; hence the measured processing intervals are realistic values for these applications. The software overhead required to service the traps could not be fully eliminated and therefore the measured communication costs are higher than on large-scale parallel systems of the future. These measurements serve to test our conclusions under high communication overhead.

We ran the simulator with a 64 kilobyte, 4-way set associative cache per node, and a full-directory non-broadcast invalidate cache coherence protocol [ASH88]. One might expect CV_C to be high on this system because some but not all remote requests have to be forwarded from the directory to a third node that will supply the updated copy of the block, and also because there could be significant queueing delays for the trap handlers on the nodes.

The data for MP3D and Water are given in Table 3.4. The communication overhead for Water is high (as expected) but perhaps only slightly higher than is likely in communication-bound applications on future parallel systems. In MP3D however, the overhead is extremely high, because of very frequent remote communication as well as extremely high cost (1.5 milliseconds) per cache miss. In practice, applications may have to restructured for less frequent communication, and MP3D can be considered an extreme case to test the conclusions of the model. Despite the high overhead, CV_C is *not significantly higher* than in the previous experiments for either program, and CV_T is still only 0.2 or less. Unless much higher CV_C is observed when more of the communication

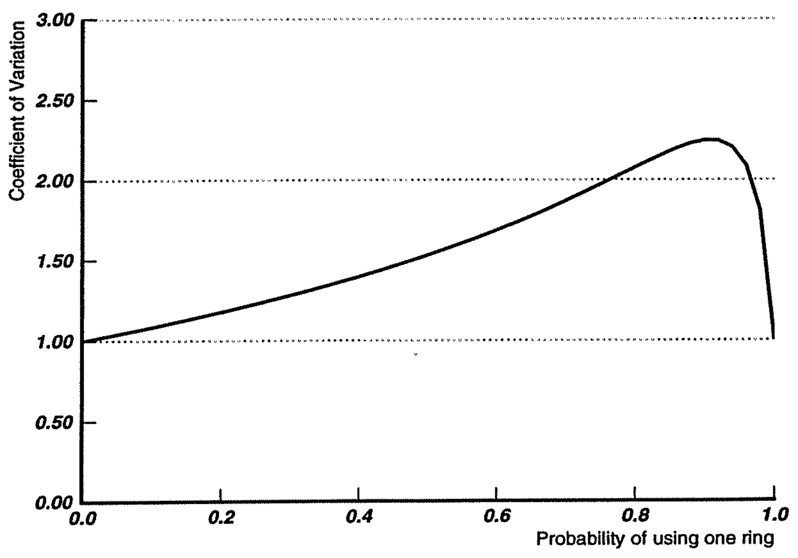


Figure 3.3. Coefficient of variation of communication costs on a 2-level ring hierarchy.

Assumptions: 1. Accesses that use both rings take 10 times as long as those that use one ring
2. Latencies in either case are exponentially distributed
(i.e., overall distribution is hyper-exponential)

is implemented in hardware, it seems plausible that even in such systems communication costs will not introduce significant variability in execution times.

Since even in the above experiments, the values of CV_C obtained were not significantly higher than on the Sequent, we explore the behavior of CV_C on another network which might be expected to have widely varying communication delays, namely the multi-level ring network such as in the Kendall Square KSR1. On a KSR1 with a 2-level network, i.e., up to 1024 processors, the *average* latencies of requests that cross both rings is about a factor of 4 higher than requests that use only one ring [KSR91]. Assume a higher ratio of 10, and assume requests *at each level* have exponentially distributed latencies (motivated by the measured values of CV_C on the Sequent as well as the CM-5). In Figure 3.3, we plot the *overall* coefficient of variation of communication latency (i.e, CV_C) as a function of the probability that a request uses only 1 ring. The figure shows that even with this network, the value of CV_C stays below 2.3. Actually, such values of CV do represent high variability in absolute terms, and these data again indicate that much higher values of CV may not occur in practice.

3.3.3. Measurements of CV_T (and $CV_{T|D}$)

The measurements we present in this subsection have two goals. First, we estimate CV_T, i.e, the variability in T due to all sources, using direct measurements of a set of the application phases, to see how much overall variability occurs in practice. Second, we use these measurements of CV_T for two purposes: (1) to qualitatively validate the values of $CV_{T|D}$ predicted by the renewal model in cases where this is possible, and (2) to compare the relative influence of variability in processing requirement D and variability in total delay C_Σ in cases where this is relevant.

Table 3.5 gives values of CV_T measured in software for several application phases on the Sequent Symmetry (estimated using samples of T from 100 to 300 runs of each phase).[10] The predicted values of $CV_{T|D}$ for these phases are repeated from Tables 3.2 and 3.3 to aid in the comparisons below. The measured values of CV_T show that in all but one of these programs, the overall variability due to processing requirements as well as communication delays is very low (less than 0.05), and it is also fairly low in the exceptional case of Locus Route.

For several of the measured phases (i.e, those in MP3D, Water and Barnes), D is fixed for different runs on the same input. In these cases, CV_T should be equal to $CV_{T|D}$, thus the measured CV_T can be directly used to validate the predicted values of $CV_{T|D}$. In each of these cases, the measured and predicted values agree within the expected accuracy of the software measurements. For the remaining cases in the table, the measured CV_T provides an (approximate) upper bound on $CV_{T|D}$. Thus, in all cases except Locus Route the measured values of CV_T directly indicate that the very low variability predicted by the renewal model is qualitatively correct.

Bicon, Locus Route and PSIM each have variability in D due to the nature of the computational algorithm. (For example in Locus Route, a process repeatedly computes the route for the next wire in a work queue, where the time to compute the route depends on which wires have previously been routed. Variability in the initial task assignments as well as in the *individual task* execution times due to random delays will cause D to vary across different runs.) In these cases, the values of CV_T and $CV_{T|D}$ (assuming these are approximately accurate) suggest that the variability in D dominates that in the total delay (but is still low). In the case of Bicon, we were able to obtain

10. A higher degree of accuracy in the measured CV_T could be obtained with increased effort by increasing the number of samples and/or by using hardware measurement probes. The increased accuracy is not needed here.

Table 3.5.. Measured Total CV_T and Predicted $CV_{T|D}$ †.

Program Phase		Input	Measured	Predicted
MP3D	*Move*	Small	0.0076	0.0020
MP3D	*Move*	Large	0.0058	0.0005
PSIM	-	Small	0.0335	0.0019
PSIM	-	Large	0.0098	0.0008
Bicon	*Connl*	Large	0.0108	0.009
Bicon	*Tour*	Large	0.0345	0.0053
Bicon	*Lowhigh*	Large	0.0404	0.0014
Locus Route	*Itn. 2*	Small	0.0882	0.0016
Locus Route	*Itn. 1*	Small	0.1396	0.0027
Water	*Inter-mol*	Small	0.0006	0.0004
Barnes	*Force*	Small	0.0008	0.0001
Barnes	*Force*	Large	0.0005	0.0000

† First case is for MP3D running on 4 processors, with bus load from 14 processors.
All others are on 16 processors, with no external load.

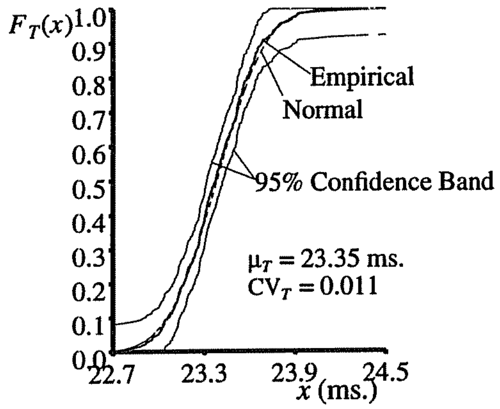
further evidence for this conjecture, using the following calculation for the *Lowhigh* phase. Each task execution time in this phase is determined by a measure of a node in the graph that appears to vary quite randomly for each task across different runs. Furthermore, the measured variance of execution times of all tasks of the phase in a single run on one processor agreed very closely with the measured variance of a subset of the tasks that would have been assigned to the measured process during a parallel run. We thus hypothesized that the measured variance in task times on a single processor is a good estimate of the variance in task processing requirements across different parallel runs, for the tasks assigned to the process. Using this to calculate the component of CV_T^2 due to variations in D, we obtained a value of $(0.042)^2$. Thus, this value added to $CV_{T|D}^2$ agrees quite closely with the measured value of CV_T^2.

3.3.4. Measurements of Execution Time Distribution

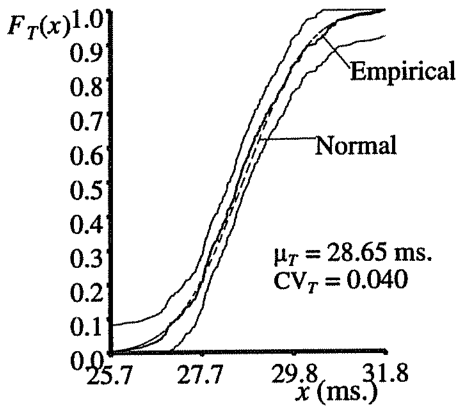
In Section 3.2.2, we showed that the distribution of execution time in an interval asymptotically approaches a normal distribution, when the non-determinism is due to random delays. Since this is an asymptotic result and since there are other sources of non-determinism, it is important to determine if real program phases show normally distributed execution times. This can be done using the measured samples of running times used in Section 3.3.3. Here, the samples are used to construct an empirical distribution which is an estimate for the unknown parent distribution. Furthermore, the mean μ and variance σ^2 of the samples can be used to construct a Normal distribution, and thus the shape of the estimated parent can be directly compared to a Normal for each phase. In addition, the Kolmogorov-Smirnov statistic can be constructed from the measured samples and used to derive a *confidence band* for the actual parent distribution [Tri82]. This gives an error bound between the estimated and actual parent distribution at a certain level of confidence.

The empirical distribution calculated using 300 samples, the upper and lower ends of the 95% confidence band, and the predicted Normal distribution, are shown in Figure 3.4(a) for the *Conn1* phase of Bicon. We see that the empirical distribution very closely tracks the normal distribution. The corresponding curves are given for the *Lowhigh* phase in Figure 3.4(b), and although we believe the variations in task execution times are the dominant cause of variance in this case, we again see that the empirical distribution closely tracks the normal. In fact, this is not surprising because the measured process executes 256 individual statistically identical tasks, and thus the sum of the processing requirements itself has converged to a normal distribution. The width of the confidence bands is ±0.076 in both cases. The same data (from 80 and 100 samples) are shown in Figure 3.4(c) for the *Move* phase of MP3D executing on 16 processors with an input size of 20000 particles, and in Figure 3.4(d) for MP3D executing on 4 processors with input size of 5000 particles, but with the external bus loading program executing on 14 processors in the latter case. In both cases, the distribution is very close to normal.

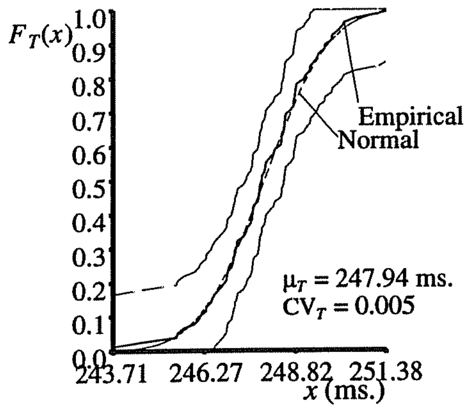
Finally, the curves for the two iterations of Locus Route are shown in Figure 3.4(e,f). The empirical distributions (calculated using 100 samples each) are different from the corresponding (predicted) normal distribution in each case, particularly for Iteration 1 which clearly shows a trimodal form. The processing demands of a task in this program can be affected by which tasks have been completed previously (and thus even by the order of previous completions), and in a few runs the execution times of the iterations are significantly higher than in most other runs. The variability in communication delay, which would otherwise yield normally distributed execution times, is much smaller than this variability in processing demand.



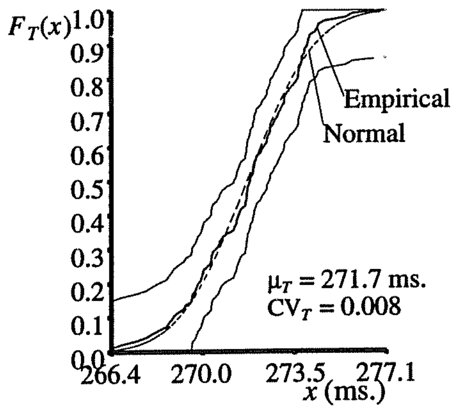
(a) Bicon: Phase *Conn1*



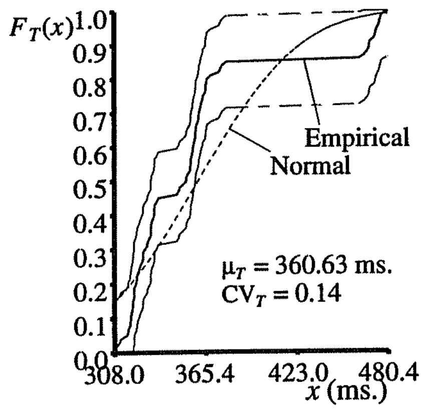
(b) Bicon: Phase *Lowhigh*



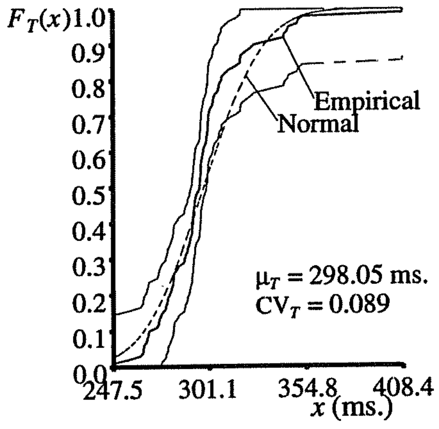
(c) MP3D: Phase *Move*



(d) MP3D: Phase *Move* (with Bus load)



(e) Locus Route: *Iteration 1*



(f) Locus Route: *Iteration 2*

Figure 3.4. Comparing measured execution time distributions and predicted Normal.

3.4. Summary and Discussion

In this chapter, we have studied the effect of random delays, as well as other sources of non-determinism, on the execution time of processes in parallel programs. We described an analytical model of program behavior that yields considerable insight into the effect of random delays on the variance and distribution of process execution time over any interval. We used detailed measurements of several shared-memory programs on two different systems to parameterize and apply the model to those programs, and thus to evaluate the variance of process execution time between synchronization points due to communication delays. We also used direct measurements of variance due to all sources of non-determinism. The key conclusions of our study are:

- In programs on current shared-memory systems, communication delays introduce negligible variance into the execution time of a process between successive synchronization points, even under conditions of high communication cost and contention. Furthermore, this conclusion should continue to hold for systems in the foreseeable future, at least for shared-memory programs with granularities similar to those on current systems.
- For many but not all shared-memory programs, non-deterministic processing requirements also introduce very little variance into the execution time between synchronization points.

These results have potentially important implications for the performance prediction of parallel programs and for general stochastic models of parallel systems. First, the results suggest that performance models of parallel programs should have the ability to represent task executions times with fairly low variance, and process execution times with extremely low variance. In Chapter 2, however, we saw that previous stochastic models that apply to any but the simplest program structures have had to assume exponentially distributed task execution times for analytical tractability. Even with this assumption, these models require extremely complex and heuristic solution techniques [KME89, MaL90, Moh84, ThB86]. It thus appears important to re-evaluate the usefulness of stochastic models for parallel program performance prediction, and perhaps to develop an alternative approach. Chapters 5, 6, 7 in this thesis explore these questions in some detail. Thereafter, in Chapter 8, we briefly discuss some other implications of the results of this chapter, including implications for parallel system performance models.

In this chapter, we also used the analytical model to prove that process execution times in the presence of random delays asymptotically approach a normal distribution, and we used direct measurements of the distribution of process execution times to show that, in practice, phases of many real programs exhibit a distribution that is very close to normal. This result, while interesting in itself, may also have some practical implications. For example, in simulation studies of parallel systems, some assumptions about the workload are necessary. In particular, when non-deterministic tasks are modeled, some default distribution is often chosen. Our results show that a normal distribution should be a reasonable choice in many cases.

Chapter 4

A Deterministic Model for Parallel Program Performance Prediction

The discussion of previous models in Chapter 2 showed that stochastic models proposed so far have not had significant proven success for performance prediction of parallel programs. Except for models restricted to simple fork-join task graphs, these models have required complex solution techniques as well as the assumption of exponential task times for analytical tractability; nevertheless, neither the efficiency nor the accuracy of these models has been demonstrated for actual programs. In fact, we believe that the limited success of these models is inherently due to the non-deterministic assumption, and more specifically, due to the difficulty of predicting average synchronization delays at synchronization points in the presence of non-deterministic task execution times. The difficulty can be traced to the combinatorially large number of execution paths that are possible for such a program. Models for arbitrary task graphs have had to take resort to Markov chains (by assuming exponential task times) to account for all possible execution paths. Models for fork-join programs have also required simplifying assumptions such as i.i.d. task times and simplified task scheduling, and even in this case, the best known solution is only asymptotically exact as the number of processors and the number of tasks per processor become large [KrW85].

In this chapter, we propose a *deterministic* model for parallel program performance prediction, in which task execution times are represented as deterministic quantities. The advantage of the deterministic assumption is that it implies a unique execution sequence for the program, and furthermore the delay at each synchronization point in this sequence can be calculated as simply the numerical maximum of the execution times of the synchronizing processes, ignoring the variance of these execution times. Thus, we are able to develop a model that is intuitive and conceptually simple, has a solution complexity of $O(N^2)$ for a program with N tasks (and common task scheduling disciplines), and applies to programs with arbitrary task graphs.

The assumption of deterministic task times is motivated by the results of the previous chapter, as we explain in Section 4.1. The deterministic model is presented in Section 4.2. First, in Section

4.2.1, we explain a basic form of the model that ignores communication and other resource contention costs. In Section 4.2.2, we present the complete model which incorporates these overhead costs as well. Section 4.3 discusses some issues regarding the implementation of the model. Finally, section 4.4 discusses issues that arise when deriving model inputs to apply the model in practice.

4.1. Motivation for a Deterministic Model

A fundamental limitation of a deterministic model is that it cannot account for the variance introduced by non-deterministic delays due to communication and resource contention. Although authors have cited such random delays as a primary argument for assuming non-deterministic task times [DuB82, KrW85], the results of the previous chapter in fact motivate the use of a deterministic performance model for shared-memory parallel programs, as explained below.

First, the study showed that in shared-memory programs with granularity similar to those on current systems, the principal effect of random delays due to communication and contention is to increase the mean execution time of each process between synchronization points, while the variance of this execution time remains essentially unaffected. This indicates it should be reasonable to ignore the variance introduced by such delays when computing synchronization costs in a performance prediction model.

Although the above result applies to overall process execution times, it indicates that it might be reasonable to represent the execution times of *the individual tasks* as deterministic quantities as well. In particular, this implicitly introduces the additional assumption that the *sequence* of task execution times is also deterministic. However, the second key result of the previous chapter showed that in many programs the total CPU requirement of a process in an interval between synchronization points is also deterministic or close to deterministic, i.e. the CPU requirement varies very little in different executions for a particular input. This at least indirectly indicates that this additional assumption may also not introduce significant errors for many programs.[11] Thus, in our model, we represent each task execution time as a deterministic quantity equal to the sum of the CPU requirements of the task and the mean total overhead experienced by the task.

11. In programs where the conclusion does not hold, i.e., if the execution time varies appreciably across different runs, the deterministic model may nevertheless provide results about one particular execution, as will become clear from the description of the model.

4.2. The Deterministic Model

Throughout this thesis, unless noted otherwise, we assume that the allocation of tasks to processes is non-preemptive, i.e., a process executes an allocated task to completion before starting on the next available task, if any. To simplify the description of the model we initially also assume that only one process per processor is used during the execution of the program, as is true in many parallel programs today. When multiple processes per processor are used, the allocation of processing power to the processes or tasks must be taken into account in the model. We will discuss how this can be done in Section 4.2.3.

The model inputs are described formally in Table 4.1. We deliberately separate the CPU requirements from other resource usage parameters (such as demand for memory modules, interconnection network resources, etc.). The number and type of the latter class of parameters will typically vary from one system to another. In the table, we have represented the task scheduling algorithm in the form of a scheduling function *Sched*(*ready_task_list*,*p*) which, given a list of ready tasks and an idle process *p*, specifies which task, if any, will be executed next by process *p*. This definition is very general, but it is not suitable for use as an input representation in practice. Instead, many common scheduling algorithms can be specified much more concisely to the model. We will discuss this issue further in Section 4.3.

Table 4.1. Inputs to the Deterministic Model .

Parameter	Explanation
N	Number of Tasks (Task 1 is assumed to be the only task with no predecessors)
Parents(i)	List of direct predecessors for each task i, $1 \le i \le N$
T_i	CPU requirements for each task i, $1 \le i \le N$
$\{M_{i,j} : 1 \le j \le N_{param}\}$	Resource usage parameters for each task i, $1 \le i \le N$
Sched(*ready_task_list*,*Idle-Proc*)	Scheduling function: specifies which task from *ready_task_list* (if any) is executed next by process *Idle-Proc*
P	Number of Processors

Figure 4.1. The Basic Deterministic Model.

Inputs

All inputs listed in Table 3.1 except resource usage parameters $\{M_{i,j}\}$.

Algorithm

```
$T_{remain}(i) \leftarrow T_i,\ 1 \le i \le N$                /* Remaining cpu requirement for task i */
$E_{set} \leftarrow$ { Task 1 }                               /* Set of executing tasks */
Do N times {
    • next_task_done ← $t \in E_{set}$ : $T_{remain}(t)$ is minimum
      Delete next_task_done from $E_{set}$
    • $T_{elapse} \leftarrow T_{remain}$(next_task_done)      /* Interval since last task completion */
      $T_{total} \leftarrow T_{total} + T_{elapse}$           /* Elapsed time since start of program */
      $T_{remain}(t) \leftarrow T_{remain}(t) - T_{elapse} \quad \forall t \in E_{set}$
    • For all immediate successors c of task next_task_done
          If all parents of c are done
              Insert c in ready_task_list
    • For each free process p
          If Sched(ready_task_list, p) finds a task t to execute
              Add t to $E_{set}$
}
Total Program Execution Time = $T_{total}$
```

4.2.1. The Basic Model Ignoring Communication and Other Resource Contention Costs

For a parallel program and a fixed input data set for that program, assume that

(1) the execution times of the tasks are each deterministic and known, and

(2) all communication and contention costs and other overheads are negligible.

Then, for a particular number of processors, this program has a *unique execution sequence*, i.e. a unique starting and ending time for every task relative to the start of the program. Furthermore, a simple algorithm can be used to compute this execution sequence, and thus the *exact* execution time of the program. This basic algorithm is outlined in Figure 4.1. Essentially, it consists of repeating the following four steps N times:

- Delete task with minimum remaining CPU requirement from E_{set} (the set of executing tasks)
- Update remaining time of other tasks in E_{set}
- Find any newly ready tasks and add to *ready_task_list* (list of ready tasks)
- For each idle process, select a ready task (if available) from *ready_task_list* according to scheduling function and add it to E_{set}

The complexity of the first three steps in the algorithm is $O(N P_{\max} + E)$ because, for the first two steps, the size of E_{set} 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 complexity of the fourth step, and hence of the overall algorithm, depends on the cost of evaluating the scheduling function. For many common scheduling functions such as typical static and dynamic task scheduling schemes (including most of those used in the applications considered in this thesis) this cost is O(1), while some more sophisticated functions such as some semi-static scheduling schemes may have a cost that is $O(n)$ for a ready-list containing n tasks. In the former case, the overall cost of computing the running time will be $O(N P_{\max} + E)$, for a graph with N nodes, E edges, and maximum parallelism $P_{\max}$. This follows because at most $N P_{\max}$ choices from the ready list need to be made in all. With a careful implementation, the same complexity can be ensured even in some cases where the scheduling function cost is $O(n)$. Specifically, this will be possible in cases where it can be detected in O(1) time that no ready task is available for a particular free process p; in such a case, 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 $O(P_{\max})$. In practice, we believe that the algorithm should be extremely efficient for any practical task scheduling method.

The above algorithm, namely computing the unique execution sequence for a program, is the essence of the deterministic model. We believe that, unlike stochastic models suggested so far, this approach is conceptually simple because it conforms well with the intuition programmers bring to the design of parallel programs. For example, in the extreme case of an unlimited number of processors the algorithm just computes the critical path in the graph, which is essentially how programmers reason about program execution time. The existence of such a simple, yet exact, underlying model is an important advantage of the deterministic approach.

Even in this basic form (i.e., ignoring overheads such as communication costs), the model can be applied to some real programs, at least on current systems, to study issues such as load-balancing and non-uniform or limited parallelism, which can influence key design questions such as task scheduling and granularity. In Section 5.3, we will use this model to predict the running times of two real parallel programs with complex and moderately large task graphs. (In fact, these programs are extremely difficult if not impossible to model with previous stochastic models.) We will also use this basic model in Chapter 7 to compare alternative program designs in one of the programs, and to obtain significant performance improvement.

4.2.2. The Complete Model Including Overhead Costs

For a model to apply to most programs, communication, contention and other overhead costs must also be represented. As explained in Section 4.1, we represent these costs as deterministic quantities that are added to the deterministic CPU requirements of each task. Thus, the total task execution times are still deterministic and, once they have been computed, the model above can be used *unchanged* to compute the overall execution time of the program. The chief remaining issue therefore is how the mean communication and contention costs should be computed and incorporated into the task execution times.

The most general stochastic models (described in Chapter 2) have typically computed resource contention costs separately for each combination of tasks in execution. As explained earlier, however, the number of such states in stochastic models grows exponentially with the parallelism in the program. A key distinguishing feature of the deterministic model is that, unlike stochastic models, it represents a unique execution sequence for the program on a particular input. This sequence will contain at most N states for a program with N tasks (exactly N states if no two tasks terminate simultaneously), where a state denotes the list of task-process pairs for all the executing tasks. Thus, assuming an efficient system-level model, it should be possible to efficiently compute communication and contention costs for every state.

The actual choice of system-level model used to calculate the communication and queueing delays is strongly dependent both on the system under consideration, and perhaps also on the required accuracy of the modeling study. Therefore, unlike many previous authors [KME89, MaL90, ThB86], we do not specify any particular queueing network framework to be used at the system level. In modeling applications on the Sequent Symmetry multiprocessor, we used a simple but highly system-specific queueing model for the bus and memory modules, incorporating a few key features such as limits on the number of outstanding memory requests. This queueing network model is described in Appendix B.

Given some model of the system, the method we use to incorporate mean delays into the calculation of execution time is as follows. We assume that the system model gives the expected total delay $T_{delay}(t, cpu\_service, cur\_state)$ experienced by task t over an interval in which it receives *cpu_service* units of service at its processor, when the system is in state *cur_state*. (In practice, only one solution of the queueing network in state *cur_state* is needed to compute $T_{delay}(t, \cdots, cur\_state)$ for all executing tasks t.) In addition to the four steps of the basic algorithm, a new step is added to the beginning of the loop (the complete algorithm is shown in Figure 4.2):

- Solve system-level model to calculate for each executing task t the total delay, $T_{delay}(t, T_{remain}(t), cur\_state)$, that it would incur until it completes execution *assuming no other task completes first* (i.e., assuming the system stays in state *cur_state*).

Note that this condition will actually be true only for the task that completes first and that task is the one having the minimum value of $T_{remain}(t) + T_{delay}(t, ...)$. This interval is denoted T_{elapse} as before. For each other task t, we assume that the total delay experienced in this interval will be $T_{elapse} \times \frac{T_{delay}(t)}{T_{remain}(t) + T_{delay}(t)}$. (Intuitively, we assume that the delay would be spread uniformly over the remaining life of the task, if the task had been the first to complete.)[12] Thus, each task t will receive an amount of CPU service equal to $T_{elapse} \times \frac{T_{remain}(t)}{T_{remain}(t) + T_{delay}(t)}$, and $T_{remain}(t)$ is updated to reflect that. Otherwise, the algorithm stays just the same as in the basic model.

The complexity of the full model solution depends on the cost of solving the system-level model, usually a queueing network in which each processor (or perhaps each executing task) is represented as one customer. Even in the latter case, the number of customers is $O(P_{\max})$. The number of queueing centers will typically be a small constant (e.g., 5 in our model of the Sequent bus and memory sub-system.) An MVA solution technique will suffice since only mean values of task delays are required; furthermore approximate customized MVA has repeatedly proven extremely accurate for queueing network analysis of multiprocessor system performance

12. "One-time" overhead costs, e.g., the cost incurred 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 and directly incorporated by delaying the start of the task in the execution sequence.

Figure 4.2. The Complete Deterministic Model Including Overheads.

Inputs

All inputs listed in Table 3.1.

Algorithm

```
$T_{remain}(i) \leftarrow T_i, 1 \le i \le N$           /* Remaining cpu requirement for task i */
$E_{set} \leftarrow$ { Task 1 }                          /* Set of executing tasks */
Initialize current state cur_state                       /* List of task-processor pairs (one per task in $E_{set}$) */
Do N times {
    • Solve system-level model to calculate $T_{delay}(t, T_{remain}(t), cstate)$ for all executing tasks t
    • next_task_done $\leftarrow$ $t \in E_{set}$ : $T_{remain}(t)+T_{delay}(t)$ is minimum
      Delete next_task_done from $E_{set}$
    • $T_{elapse} \leftarrow T_{remain}$(next_task_done) + $T_{delay}$(next_task_done)
                                                         /* Interval since last task completion*/
      $T_{total} \leftarrow T_{total} + T_{elapse}$       /* Elapsed time since start of program */
      $T_{remain}(t) \leftarrow T_{remain}(t) \times \left(1 - \frac{T_{elapse}}{T_{remain}(t) + T_{delay}(t)}\right) \; \forall t \in E_{set}$
                                                         /* Subtract part of $T_{remain}(t)$ completed in time $T_{elapse}$ */
    • For all immediate successors c of task next_task_done
          If all parents of c done
              Insert c in ready_task_list
    • For each free process p
          If Sched(ready_task_list, p) finds a task t to execute
              Add t to $E_{set}$
      Update current state cur_state                     /* Delete task next_task_done and
                                                            add newly started tasks */
}
Total Program Execution Time = $T_{total}$
```

[AdV93, VLZ88, WiE90]. The solution complexity of approximate MVA is relatively insensitive to the number of customers (indirectly, the number of iterations required for convergence can be affected by the customer population). Finally, from a single solution of the queueing network, the mean response times and thus the mean total delay $T_{delay}(t, \ldots)$ can be computed for all $t \in E_{set}$ in time proportional to the size of E_{set}, which is $O(P_{max})$. Thus, the complexity of this step is $O(P_{max})$, i.e. it contributes $O(N\,P_{max})$ to the complexity of the overall algorithm. Thus, the overall complexity is the same as for the basic model. (Exceptions to the above arguments arise only if a *non-homogeneous* queueing network is used and solved using *exact* MVA, or if some other solution technique is used that has cost *greater* than O(n) with n executing tasks.)

In practice, however, the system-level solution step could easily dominate the overall solution time of the model, and reducing the number of times the system-level model must actually be solved is invariably worthwhile. A suitable method that is often extremely effective is discussed in Section 4.3.

Finally, a desirable and potentially useful property of the (basic or full) deterministic model is that it gives exactly the same results whether used with the original task graph or the condensed graph, in programs for which the condensed graph can be constructed. This is easiest to see in the case of the basic model: when tasks are statically allocated to the processes, the execution sequence will be exactly the same whether tasks that would be condensed into a single node are enumerated one at a time, as with the original graph, or all together, as with the condensed graph. Since the CPU requirements time of a node in the condensed graph is just the sum of the individual CPU requirements of its component tasks, the model gives identical results in both cases. With the full model, this will continue to hold if tasks that are condensed into a single node have identical resource usage parameters. It then follows from the above uniform delay assumption that the model will compute the same total delay time for the interval in which a processor executes the tasks of a condensed node, whether the tasks are specified separately in the original task graph or as a single condensed node in the condensed graph. Our experiments with statically scheduled programs (discussed in Section 4) have borne out these conclusions.

4.2.3. Extensions to the model for multiple processes per processor

So far we have assumed that a single process per processor is used during the execution of the program, as is true in many parallel programs today. When multiple processes per processor are used, the allocation of processing power to the processes must be taken into account in the model. This will typically require system-specific modifications to the model. We use two examples to illustrate the changes that could be required.

First, consider the case where idealized processor-sharing is used to schedule the N_p processes of the program on $P \leq N_p$ processors, but processes do not relinquish the processors when accessing other shared resources, such as for remote communication. Then, referring to Figure 4.1, the only required change in the basic model is that the value of T_{elapse} is now calculated as

$T_{elapse} \leftarrow T_{remain}(next\_task\_done) \times \min\left\{1, \frac{P}{\min\{\,|E_{set}|, N_p\,\}}\right\}$, where $|E_{set}|$ gives the number of executing tasks. In the full model (Figure 4.2), in addition to this change, the solution of the system-level model must also take into account that each of the P processors (customers in the queueing network) is "simultaneously" executing multiple processes with possibly different resource usage behavior. Otherwise, the models remain unchanged.

When processes do relinquish processors for remote communication or access to other shared resources (note that this would only be useful if there were more processes than processors), an even simpler modification suffices. Possible examples of such systems are multi-threaded processors [ALK90] or parallel applications with significant I/O requirements. Representing such a program with the model only requires modifying the system-level queueing network to represent the individual processes instead of the processors as customers, and to include the P processors as individual queueing centers with some appropriate scheduling discipline. Otherwise, the algorithm of Figure 4.2 remains unchanged.

4.3. Model Implementation Issues

4.3.1. Optimizations

Some care in certain aspects of model implementation can make the model solution significantly more efficient. One fairly obvious technique for this purpose has the additional benefit of simplifying the input representation of the task graph. Specifically, tasks with a common set of predecessors and successors as well as common values of resource usage parameters can be specified and operated on as a single *task group*. Only the task CPU requirements need be specified and stored separately for the individual tasks in the group. This method is particularly useful for programs with loop-based parallelism which often have very large numbers of iterations per loop, all of which can be represented as a single task-group leading to dramatic savings in the size of the input specification and also some savings in model solution time. For example, updating the ready-list (step 3 or 4 in the basic or full model) only needs to be done when an entire task group completes execution.

Second, large savings in model solution time can be obtained for many programs by avoiding unnecessary solutions of the system-level model as far as possible. Specifically, that model only needs to be solved when the total number of executing tasks changes or a newly started task has different resource usage parameters from the task that just completed. For example, in the loop-based programs mentioned above, large groups of tasks may have identical resource-usage requirements and typically only one such group is executing at a time. Therefore, the number of required solutions of the system-level model can be reduced by an order of magnitude or more.

4.3.2. Specifying the Task Scheduling Function

While the form of the task scheduling function in Table 4.1 is quite general, it is not appropriate for use as an input specification in the implementation of the model: clearly, it is impractical to list the values of the function *Sched*(*ready_task_list*, $\cdots$) for every combination of tasks in *ready_task_list*. Instead, a common scheduling framework that includes a broad class of common task scheduling functions is provided in the deterministic model implementation. (The framework is based on a few simple primitives in the code, specifically, one or more task queues that together contain all the tasks in *ready_task_list*, along with routines to insert or delete tasks in these task queues.) The most common task scheduling functions including static scheduling of groups of tasks (usually a loop) or dynamic task scheduling based on a single task-queue can be directly specified as single command-line options. In addition, a large class of static and semi-static scheduling policies that are also represented in the framework can be specified fairly concisely in an input file. (The programs PSIM and DynProg described in Section 5.1.1, and the scheduling function for Locus Route studied in Section 7.3 provide examples of such policies.) However, some types of scheduling methods are inherently difficult to support in a generic model implementation. In particular, the scheduling algorithm can involve detailed computation as part of the program, perhaps considering many details of the input values (for example, task allocation based on the evolving spatial distribution of objects in an N-body problem). To generate the input specification of scheduling for any task-graph based model, some portion of this computation might have to be reproduced. Whether this is possible and how much effort is entailed would depend heavily on the specific program.

4.4. Deriving Model Inputs

Of the input parameters for the deterministic model listed in Table 4.1, the key input is the task graph. Constructing the specification of the task graph (N and *Parents*(i), $1 \le i \le N$) for a program is equivalent to reproducing the parallel control structure of the program, and can usually be done from a basic understanding of the program, by carrying out little or none of the actual computation.[13] The control structure can be as simple as a parallel loop (i.e., a fork-join task structure) or can be more complex as in programs with non-series-parallel task graphs. However, even for the program with the most complex task graph we have studied, namely the program Polyroots described in Section 5.1, it was not difficult (even without assistance from the authors of the code) to write a script to generate the task graph by exploiting basic regularities in the graph structure.

The other inputs required for the model are the task scheduling function, the task CPU times, and (for the full version of the model) the shared resource usage parameters. For any scheduling algorithm, the principal question is in what form the algorithm can be specified to the model, and the issues involved have been discussed above. The methodology used to derive the CPU times and shared resource usage parameters will depend not only on the required precision of the experimental results but also on the nature of the experiment. In particular, the approach to deriving these parameters depends on whether the model is being used to evaluate an existing program or to evaluate potential program design changes. Below, we discuss the issues involved in obtaining model inputs in each of these two cases.

4.4.1. Deriving Model Inputs for Evaluating an Existing Program

In current practice, performance studies most commonly focus on evaluating an existing program on an existing system. For a given program and input, the CPU requirements of the individual tasks, $T(i), 1 \leq i \leq N$, can be obtained by direct measurement. An advantage of our definition of a *task* in Table 2.1 is that these values can be measured from a single execution of the program, and then used for all analyses of this program for this input. The most convenient method is to measure these values directly in software using explicit system timers or using software instrumentation tools such as pixie and qpt [BaL92]. Note that some software timing techniques will implicitly include overheads due to communication or shared-resource contention, rather than purely the CPU requirements. Depending on the desired accuracy, this effect could be simply ignored, or some degree of care could be used to minimize the effect during measurements (such as measuring while executing stand-alone on 1 processor). If even greater precision is desirable, such as in our model validation

13. In exceptional cases such as parallel discrete event simulation, much of the actual computation would have to reproduced to generate the task graph.

experiments, these additional overheads could be estimated and subtracted from the CPU requirements. (The overheads can be easily estimated from the shared resource usage parameters, obtained as described below.)

It is also desirable to be able to study the performance of a particular program on a range of inputs, rather than a single input, and to do so efficiently. In measurement or simulation-based performance tools, this typically requires a new execution for each case. A more abstract model such as the deterministic model, however, provides greater flexibility. In particular, in algorithms where the number of tasks and the computational requirement of each task scale as simple functions of one or a few input parameters, it will be possible to study a range of program inputs by extrapolating from a single set of measured task CPU times to obtain the necessary model input values for each case. A common example occurs in parallel loops where the number of iterations and the CPU requirement per iteration each scale as some function of one or a few input parameters. Note that it may not be necessary for this scaling to be extremely precise. Standard algorithm analysis techniques for sequential algorithms could be used to determine both how the CPU requirements scale, and how accurate the scaled estimates might be. For example, the application PSIM, a multistage network simulator, (studied in the next section) contains a number of parallel loops. The number of iterations in each loop is proportional to K^N (where K and N are the base and order of the simulated network); the CPU requirement of each iteration is constant in some loops, while in others it is another simple function of N and K such as $N K^2$. Thus, once the CPU requirements for one value of N and K are measured, the performance for other combinations of N and K can easily be studied. In fact, similar accurate scaling of the number of tasks and the task CPU times is possible in 3 of the 5 applications studied in the next section.

When using the full deterministic model, communication and other shared resource usage parameters ($\{M_{i,j}: 1 \le j \le N_{param}\}, 1 \le i \le N$) must also be obtained. The deterministic model does not prescribe a particular set of parameters since it does not assume a particular system-level model, thus providing flexibility in the level of detail at which communication and shared resource contention are represented. More detailed parameters can allow a more precise representation of communication behavior, but obtaining the parameter values would require greater effort. For example, on the Sequent Symmetry, the cache miss rate alone provides useful information about the communication behavior of an application, and can be estimated directly from software by reading special hardware counters that monitor cache misses per processor [Seq81]. Additional parameters such as the fraction of misses that are satisfied by a remote cache instead of by main memory, or the fraction of cache misses that require write-backs, allow a more accurate prediction of the mean response times on cache misses, but also require more detailed measurements. An additional limitation of using detailed system-level parameters is that the parameter values obtained typically will not generalize to other system configurations or to other systems. More abstract or high-level parameters describing the inherent communication in the program could provide greater flexibility for predicting performance across a range of system configurations.

4.4.2. Deriving Model Inputs for Studying Program Design Changes

In a parallel program performance study, it is important to be able to evaluate the effect of program design changes or changes to the underlying system, and it is desirable to be able to do so with as few additional measurements as possible. Whenever possible, it is especially attractive to be able to do so before actually having to implement such changes. By using the deterministic model, some important kinds of program or system changes can be fully or partially explored without requiring *any* additional measurements (and thus without requiring the changes to be implemented). In particular, if it is reasonable to ignore any changes in shared resource usage parameters such as cache miss rates, then issues such as the effects of system changes on mean communication latencies or the effects of changes in task scheduling (i.e, granularity and task allocation) on overall performance can easily be studied without additional measurements. For example, analyzing changes in task scheduling for a given program only requires specifying a new scheduling function, without changing the task graph or measuring new values of the task CPU times or the task resource usage parameters (when the latter remain approximately fixed). This predictive power of the model is principally due to the abstract task graph representation of parallelism, together with the appropriate identification of the tasks (i.e., conforming with the definition in Table 2.1), the separate representation of task scheduling, and the separate system-level model of shared resource usage. Even if changes in task scheduling do effect the low-level parameters such as communication rates, it may still be possible to study the approximate impact of such changes on load-balancing, without requiring new parameter values. In Chapter 7, we provide a few examples of using the model to analyze these various kinds of design changes.

For program or system changes that significantly affect low-level parameters such as mean communication rates, predicting the full performance impact of such changes requires estimating or measuring the new values of these parameters. In some programs, sequential algorithm analysis techniques may provide at least rough estimates of how mean communication requirements (for example) scale with input size and the number of processors used, and the effect of alternative scheduling disciplines may be analyzable as well. In fact, for algorithms with simple, regular and fixed data-reference patterns, Tsai and Agarwal have shown that it is possible to derive precise analytical estimates of multiprocessor cache miss rates as functions of input size, cache block size, and the number of processors [TsA93].

Finally, when using a model in early stages of the design and development of a program, important parameter values, in particular CPU requirements and perhaps rough communication rates, may have to be estimated or measured from partially developed code. The abstract representation of a program using tasks and a task graph should prove helpful in understanding what pieces of the algorithm are necessary to isolate and measure.

Chapter 5

Comparative Evaluation of Deterministic and Stochastic Models

In Chapter 4, we described a conceptually simple deterministic model for parallel program performance prediction. In this chapter, we will evaluate the efficiency and accuracy of the deterministic model, and compare the deterministic and stochastic approaches for parallel program performance prediction. For this study, we use five shared-memory programs on a Sequent Symmetry. The programs are described in Section 5.1, along with the methodology used for the study. Three of the five programs have fork-join task graphs, and in Section 5.2 we use these three programs to evaluate the full deterministic model as well as representative stochastic models, and thus to compare the two approaches. The two remaining applications have more complex, non-series-parallel task graphs, and the only previous stochastic models that apply to these programs are the three Markov chain models described in Section 2.2. However, the exponential time and space complexity of these models make it impractical to use them to analyze these two programs. (Our results in Section 5.2 will corroborate this claim.) In contrast, the deterministic model can easily be applied to these programs, and we do so in Section 5.3 to further demonstrate the efficiency and accuracy of this model. In fact, these two programs have very low communication and contention overhead, at least on the Sequent. Hence, we will apply the *basic* deterministic model, which ignores overheads, and thus demonstrate that even this extremely simple model can be quite useful in practice.

5.1. Applications and Methodology Used in the Study

5.1.1. Applications

The five programs used in this study are briefly characterized in Table 5.1. The key common features of the applications are that all five are scientific and engineering applications written for shared-memory systems, they are written assuming one process will be used per processor during execution, and they do not have significant I/O requirements. Three of the programs (MP3D, PSIM and Locus Route) were briefly described in Section 3, where they were used in the study of

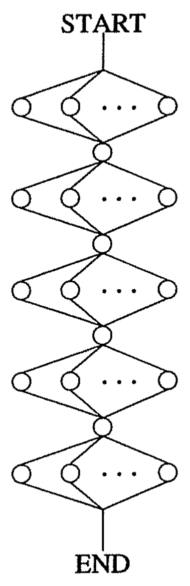
Table 5.1. Applications Evaluated Using the Analytical Models.

Name	Application Description	Task Graph Structure	Task Allocation Method	Cause(s) of Perf. Loss
Poly-roots	Compute roots of 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 to processes	Limited parallelism at beginning and end
MP3D	Particle simulation in rarefied fluid flow	Fork-join: five parallel loops per iteration; one loop contains more than 90% of total work.	Static allocation of loop iterations in each loop.	Cache misses; 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 a few large tasks; cache misses

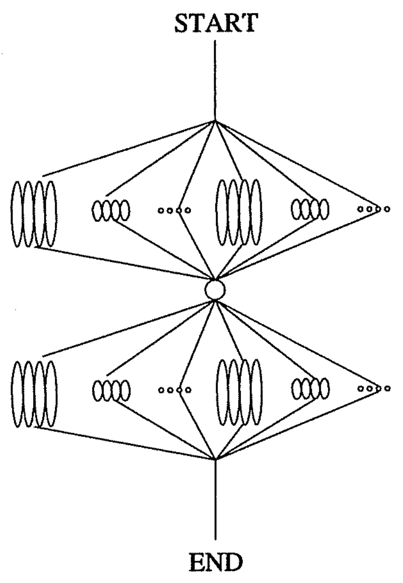
random delays. In this table and the description below, we discuss in a little more detail the characteristics relevant to this study, particularly the task graphs, scheduling methods, and the nature of variations in the task times.

The first program, MP3D, is taken from the SPLASH suite of parallel applications [SWG92]. It simulates the motion of particles in very low density fluids. The task graph for this program is shown in Figure 5.1(a). It is an iterative fork-join task graph with five parallel phases (parallel loops) per iteration. In each parallel loop, chunks of 8 consecutive loop indices are always allocated as a single unit and hence we can consider a chunk to be a single task (see Table 2.1). There are small but significant variations in the task times in each parallel loop. The tasks are statically allocated to the processes in cyclic order (one of the cases supported directly in the deterministic model implementation).

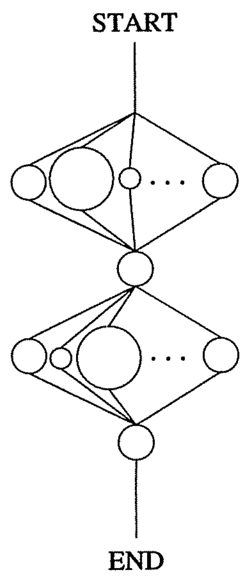
PSIM is an interconnection network simulator written in PCP, a parallel extension of C that supports efficient nested forking within programs [Bro88a]. PSIM is a fork-join program in which each parallel phase consists of 6 parallel loops with no intervening barriers (Figure 5.1(b)). The



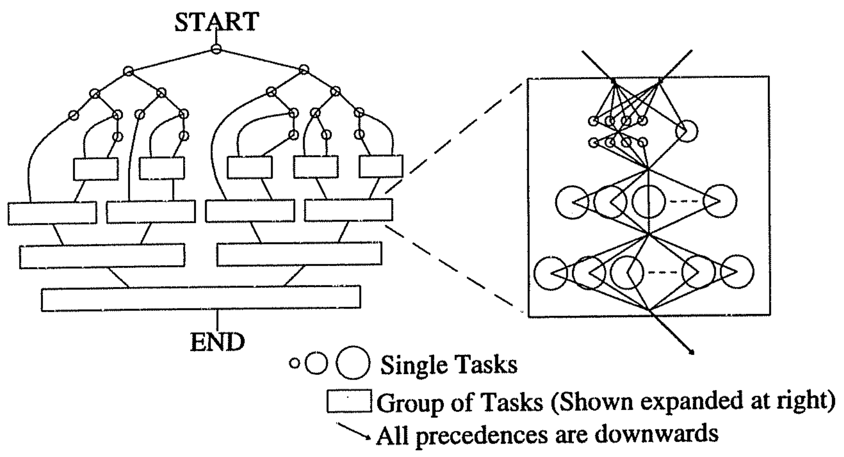
(a) MP3D (One Iteration)
5000 particles (539 tasks)
20000 particles (1978 tasks)



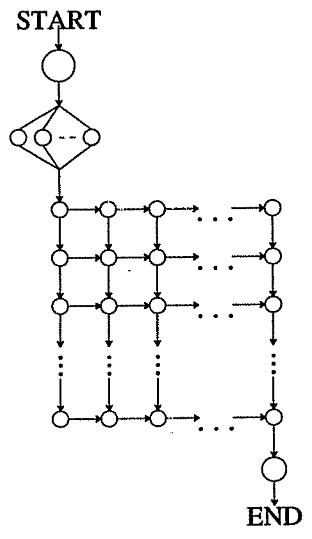
(b) PSIM (One Iteration)
1024-node system (10243 tasks)
4096-node system (40963 tasks)



(c) Locus Route
Circuit bnrE.grin
(843 tasks)


START
END
Single Tasks
Group of Tasks (Shown expanded at right)
All precedences are downwards

(d) Polyroots
Polynomial Degree 20 (217 Tasks)
Polynomial Degree 30 (348 Tasks)



(e) DynProg
Sequence Length: 100 (1403 Tasks)
Sequence Length: 500 (32003 Tasks)

Figure 5.1. Task Graph Structures of the Applications.

tasks of each loop, which correspond to individual loop iterations in this case, 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). The granularity of work per task (loop iteration) is much larger in some loops than in others. (For example, 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.) Despite this skew, the load is fairly well balanced on an *even* number of processors 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 executing on an *odd* total number of processors since the even numbered processors have to accomplish a larger amount of work in this case.

Locus Route, also a SPLASH application [SWG92], 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 5.1 (c)). One of three levels of task granularity can be chosen by the user; we examine the coarsest granularity, namely one wire per task, 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 in each iteration are highly skewed. 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 granularity are also available in the program. For the experiments in this section, we used the dynamic task scheduling from a single FIFO task queue. (We ignore the small costs associated with retrieving tasks from the task queue such as contention for the task queue lock. Experiments with the next program, Polyroots will be used to explore modeling such costs.) The other scheduling option is a semi-static scheduling method in which geographic areas of the VLSI chip are allocated to specific processors. We will use the deterministic model to evaluate the performance of this scheduling option and variants thereof in some detail in Chapter 7. Finally, an important characteristic of Locus Route is the non-deterministic nature of the CPU requirements, as observed in Section 3.3. This effect has to be taken into account when evaluating model accuracy. The effect will be quantified and discussed further when presenting the results for this program.

The remaining two programs, Polyroots and DynProg, have non-series-parallel task graphs. Polyroots computes the roots of a polynomial with arbitrary-precision integer coefficients. The task graph of this program is fixed for a particular input polynomial degree, and is shown in Figure 5.1(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 Locus Route. Scheduling overhead including contention for the task queue lock is small in this program as well. However, by introducing an artificial delay into the corresponding critical section, we were also able to study the accuracy of modeling lock contention.

DynProg, uses a pipelined dynamic programming algorithm for aligning two gene sequences. The program has a pipelined task graph (Figure 5.1(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 pipelined task array are allocated to the same processor; rows are statically allocated to processors in round-robin fashion. This scheduling method was specified explicitly to the program that solves the deterministic model.

5.1.2. Methodology For Applying the Deterministic Model

The above applications were evaluated on a 20-processor Sequent Symmetry S-81. The methods used to derive the required input parameters for the basic and full deterministic models are similar to those described in Section 4.4. In particular, scripts to generate the task graph for each of the above programs were not difficult to develop with only a basic understanding of the code. The specific scheduling functions in the programs were specified as explained above. The measurements of task CPU requirements were made from software using the hardware 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 MP3D, PSIM and Locus Route, which have significant communication overhead, we subtracted the mean communication costs on 1 processor from the measured CPU requirements since communication behavior including contention would be represented separately and precisely in the full deterministic model.

For the applications with significant communication overhead, we used a detailed set of system-level parameters to describe the communication behavior. These parameters are defined and explained further in Appendix B (along with the queueing model used to compute mean response times for communication events). To allow precise evaluation of model accuracies, the parameter values were obtained using careful hardware measurements. Since measuring these parameters for every task in a program is clearly impractical, we measured the average values of these parameters over the duration of a phase and used these values for all tasks in the phase. We also assumed that the communication behavior would stay approximately constant for 2 or more processors, but that the behavior on 1 processor might be different.[14]

Finally, 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 the above measurements were made when no other user programs were actively using the system.

5.1.3. Methodology for Evaluating and Comparing the Stochastic Models

The numerous complex heuristics used in many of the stochastic models make it impractical to implement and explicitly evaluate every model of interest. Furthermore, the exponential time and space complexity of the three Markov chain models (Thomasian and Bay, Mohan, and Kapelnikov et al.) make these models impractical for realistic task graph sizes in the programs we study (see Figure 5.1). Thus, four models were implemented and directly compared in this study: the deterministic model, 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 deterministic (full or basic), *ML*, KW_{actual} and KW_{exp} respectively.

14. We tested this assumption for the mean communication rate in two programs (MP3D and PSIM) on the Sequent and found it to be quite accurate for these two cases. For example, for the dominant phase of MP3D, 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. Although this is encouraging, the assumption may be more approximate for or may not extend to programs on larger systems.

Before describing the methodology used to evaluate and compare the stochastic models, it will be useful to understand some key characteristics of the relevant models, for a few reasons. First, these characteristics influence the methodology we used to evaluate the stochastic models. In fact, they also show how and when we can infer important aspects of the accuracy of the three Markov chain models from the results of the stochastic models that were directly evaluated. Finally, these characteristics are fundamental in determining the accuracy of the various models and therefore will be important in understanding the experimental results later in this section. Nevertheless, these characteristics have not previously been discussed even for the individual models.

The accuracy of an overall model is determined both by the accuracy of the high-level model component, as well as the accuracy of the low-level model component. However, as discussed in Chapter 2, for any overall model and for any particular system, various choices of system-level model are possible and an appropriate one can be chosen to provide the desired accuracy in the low-level model results. The more significant differences between the various modeling approaches lie in how the high-level behavior of tasks and processes are represented. Thus, in the discussion below, we focus on the accuracy of the high-level model component in each model.

The accuracy of the high-level model component depends primarily on how accurately synchronization costs are estimated, which in turn depends on how accurately the distribution of execution time of each process between synchronization points is modeled.[15] (Note that these process execution times are implicitly represented in each model, and not explicitly computed.) The representation of process execution time is determined by the combination of two key factors: the individual task execution times, and task scheduling. Thus, the accuracy of the computed synchronization costs is determined by how faithfully the combination of these two factors is represented in the model.

The stochastic model of Kruskal and Weiss allows arbitrary variance of task times, and thus the variance in each task's execution time caused by communication and contention delays can be represented precisely (if it can be estimated or measured). Nevertheless the representation of task execution times in the model can be imprecise because the model assumes all task execution times

15. Although synchronization costs depend on the *distribution* of process execution time and not just the variance, the variance is a good indicator of synchronization cost in "well-behaved" distributions, particularly when the number of processes is not extremely large. Thus, in this discussion we use the representation of variance in these models as a measure of the accuracy of the distribution.

have equal mean and variance; in particular, the differences (skew) in task CPU requirements must be stochastically represented as one component in the variance, *in addition to* the component due to communication and contention delays. Furthermore, the simple representation of task scheduling (namely, that tasks are scheduled in fixed-size batches from a single task queue) may also be imprecise for some programs. In fact, more detailed representations of scheduling are precluded by the common mean and variance assumption. For example, the model cannot distinguish between different orderings of the same set of tasks in a task queue. Within the limitations of these assumptions, however, the model should be accurate because the actual variance introduced by communication and contention delays is included.

The deterministic model removes the above simpiifying assumptions and also applies to arbitrary task graphs by assuming deterministic instead of stochastic task times. In this model, individual mean CPU requirements as well as individual mean overhead costs are represented precisely (using a set of deterministic values), and task scheduling is also represented precisely, while only the variance in the individual task times is ignored. Because it ignores the variance, the model effectively assumes that the execution time of a process between synchronization points is deterministic when computing synchronization costs. The model should be fairly accurate for programs where the variance of execution time of each process is small, as explained in Chapter 4.

The other stochastic models also remove the assumptions of the Kruskal and Weiss model and apply to non-fork-join programs, by assuming exponential task times for analytical tractability. The three Markov chain models allow different mean task times, and also represent task scheduling precisely. Because of the exponential task assumption, however, they implicitly represent the execution time of a process between synchronization points as a sequence of exponentially distributed intervals. Roughly, the larger the number of tasks executed by a process in this interval, the closer to deterministic the implicit representation of total process execution time will be. Thus, for programs with low variance of process execution time, these exponential task models could be accurate if the number of tasks per process is "sufficiently large", but could be inaccurate otherwise.

The Mak and Lundstrom model is also an exponential task model and also allows unequal mean task times but, unlike the Markov chain models, it does not consider task scheduling at all when computing synchronization costs in the high-level model, as explained in Section 2.3. Instead, in a parallel phase with N tasks, it computes the total phase execution time as the maximum of N independent, exponentially distributed task residence times. Thus, the model is likely to overestimate synchronization costs when $N > P$, as is often true. In this case (which includes all of the examples we study), it should be more accurate to use the condensed graph as input to the Mak and Lundstrom model than to use the original task graph, since the task scheduling *would* be represented precisely when creating the condensed graph. (In any case, our experimental results discussed in Section 5.2 show that the concomitant reduction in the size of the graph is essential to be able to solve the *ML* model in practice.)

Note that assuming a condensed graph eliminates the key difference between the *ML* model and the three Markov chain models, namely the representation of task scheduling. Thus, these four models should have the same accuracy when used with the condensed graph (as explained in Section 2.3), allowing us to infer the accuracy of the three Markov chain models in this case.

To understand the accuracy of the Markov chain models when used with the original task graph we can use the results of KW_{exp} model, at least for certain fork-join programs. (This was one key motivation for including the KW_{exp} model in our study.) To understand for which programs such an inference is possible, recall that two assumptions in the Kruskal and Weiss model are not present in the Markov chain models: the simplified task scheduling and equal mean task times in a phase. Because of the latter in particular, the variance of the exponential distribution in the KW_{exp} model must represent the skew in mean task execution times in addition to the variability due to communication and contention delays. Thus, when (1) the skew in mean task execution times is small compared to the variance of the exponential distribution, and (2) the scheduling of tasks in KW_{exp} is a faithful representation of the actual task scheduling, the KW_{exp} model should give results similar to the three Markov chain models.

Another advantage of including the KW_{exp} model is that it is perhaps the simplest model to use in practice. In particular, it only requires knowledge of the mean task time across all the tasks in a phase; it does not require the *individual* mean task times or the variance of task times. The results for this model will indicate whether it is sufficient to use the exponential distribution instead of the more detailed representations.

The inputs for the stochastic models evaluated here were derived from the inputs measured for the deterministic model (see Section 5.1.2). The input specification required for the Mak and Lundstrom model is essentially equivalent to that used for the Deterministic model and can be easily derived from the same parameters, at least for programs with static scheduling.[16] The Kruskal and Weiss model requires estimates of μ_{task} and σ^2_{task} (the common mean and variance of task times) in each phase of a fork-join program. We compute these for a given phase as follows. Let μ_C and σ^2_C denote the mean and variance of communication costs experienced by each task in the phase, and let $\mu_T = \frac{1}{N} \sum_{i=1}^{i=N} T(i)$. Then μ_{task} and σ^2_{task} were calculated as $\mu_{task} = \mu_T + \mu_C$ and $\sigma^2_{task} = \frac{1}{N-1} \sum_{i=1}^{N} (T(i) - \mu_T)^2 + \sigma^2_C$. Of the parameters in this expression, $T(i)$ are the task CPU requirements measured for the deterministic model (we continue to assume that these are deterministic). μ_C was calculated using the same queueing network as used in the Deterministic model. σ^2_C, however, is extremely difficult to estimate or measure directly, even with our available setup for direct hardware measurements. Instead, we used the *analytical model* described in Chapter 3 to estimate σ^2_C analytically (just as was done to estimate $CV_{T|D}$ for the study of the influence of random delays). Recall, however, that using that model in turn required detailed hardware measurements of the mean and variance of bus inter-request times and response-times. While we include σ^2_C for completeness in the model validations, this term is typically much smaller than the component of σ^2_{task} used to represent the unequal values of $T(i)$, and has a small effect on the results and accuracy of the KW_{actual} model. This step would probably be unnecessary if the model were used in practice, except for programs with extremely small tasks, which could have significant values of σ^2_C.)

5.2. Evaluation of Deterministic and Stochastic Models for Fork-Join Task Graphs

In this section, we evaluate the accuracy and efficiency of the deterministic model as well as representative stochastic models, by applying each of them to the first three programs in Figure 5.1, i.e., the programs with fork-join task graphs. We begin by assessing the efficiency of the various models. For these three programs and for the input sizes we used, the task graphs ranged from 539 tasks to 40963 tasks. Even for the *smallest* task graph, the *ML* model required over 56 megabytes of

16. Their model does not separately represent task CPU requirements and task scheduling; instead these must be combined and specified in the form of mean resource demands for the processors, to be used in the system-level queueing network model. Except for static task scheduling algorithms, these CPU resource demands would be difficult if not impossible to compute, for the same reason that the condensed task graph is difficult or impossible to compute.

memory, and about 12 minutes of execution time on a workstation with sufficient memory; larger task graphs far exceed the available memory. While some improvement may be possible with a carefully optimized implementation, even this will quickly become impractical for larger graphs because of the $O(N^3)$ and $O(N^2)$ time and space complexity of the algorithm. Thus, for all other cases, the only way we were able to apply the *ML* model was to use the condensed task graph, which is typically much smaller than the original graph for $P < 20$, the maximum number of processors available on the Sequent Symmetry. (Recall, however, that condensing the task graph is only possible for statically scheduled programs. We were therefore unable to apply the ML model to evaluate Locus Route.) Finally, the models of Thomasian and Bay, Mohan, and Kapelnikov et al are significantly more complex 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, and only if the condensed graph contains on the order of a hundred or fewer tasks.

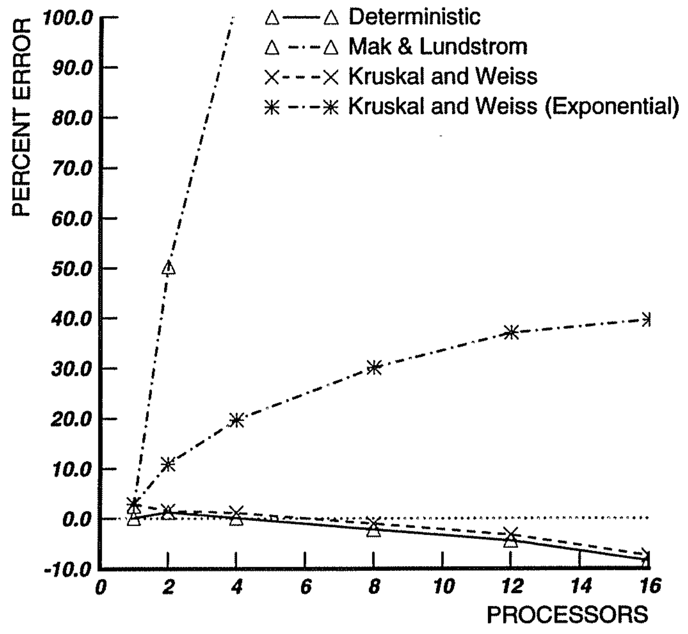
In contrast to the *ML* model, the deterministic model proved extremely efficient for all programs to which we have applied it. Even for the task graph with 40963 tasks (the largest studied in our work), the model could be solved in under 9 seconds on a DECstation 5000/125, and required less than 2.6 megabytes of memory. Finally, solution efficiency is clearly not an issue for the Kruskal and Weiss model.

The accuracy of the various models for each of the three programs are described below.

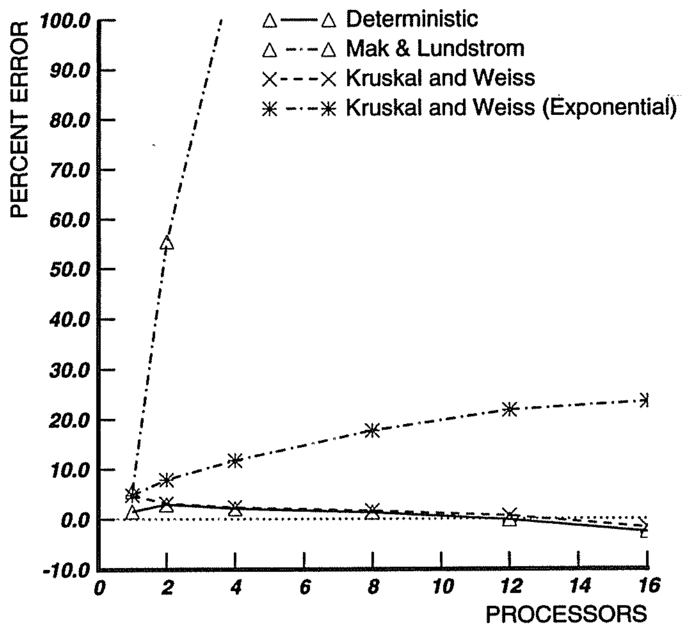
5.2.1. Results for MP3D

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 program has 539 tasks and 1978 tasks for these two input sizes respectively. The condensed task graph for 16 processors has 76 tasks for either input size (more generally, it has at most *P* tasks per parallel phase). The percentage errors in the predicted execution time from each model for different values of *P* are given for the two input sizes in Figure 5.2 (a) and (b) respectively.

The KW_{actual} model is very accurate for this application because the model scheduling assumptions faithfully represent the static loop scheduling in MP3D, and because the variations in task CPU requirements in each parallel phase are relatively small and can be adequately represented, together with the variance due to communication delays, in a single common variance parameter for each phase.



(a) Input Size: 5000 particles



(b) Input Size: 20000 particles

Figure 5.2. Errors in the Predicted Running Times for Program MP3D.

The Deterministic model is also highly accurate for MP3D, indicating that ignoring the variance due to communication delays, while representing the individual task CPU times together with the individual mean communication delays by a set of deterministic quantities, is reasonable for this program.

The error in these two models 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.

The results for the KW_{exp} model are quite inaccurate for MP3D, and the comparative accuracy of the KW_{actual} model shows that the error is due to the exponential task assumption. Furthermore the three Markov chain models, namely Thomasian and Bay, Mohan, and Kapelnikov et al, will have errors very similar to the KW_{exp} model for this program. This follows because MP3D satisfies the conditions under which these models are roughly equivalent: the scheduling in the Kruskal and Weiss model is accurate, and the standard deviation of task CPU times (e.g., 6% of the mean for the dominant phase with the larger input size) is much smaller than that of the exponential distribution. Thus, the exponential task assumption is inaccurate for this application whether it represents both variations in CPU times and variations in communication delays, or only the latter of these two quantities. In either case, the exponential assumption overestimates the variability in task execution times, leading to significantly higher synchronization costs, even though each process executes a fairly large number of tasks between synchronization points (about 100 tasks per process in the dominant phase, for the larger input size).

The pessimistic assumption of the exponential distribution is even more severe when the models are applied to the condensed graph, as shown by the results for the *ML* model. Furthermore, these results are approximately the same as would be obtained by applying the KW_{exp} model or one of the Markov chain models to the condensed graph, as explained in Section 5.1.3.[17] Thus these results indicate that using the condensed graph to allow efficient solution of large programs may not be a viable alternative for these exponential task models. (For *ML*, the error with the original graph

17. In fact, we tested this claim by using the KW_{exp} model with the condensed graph. The error curve we obtained was almost indistinguishable from the error curve for the *ML* model shown in Figure 5.2, for both input sizes.

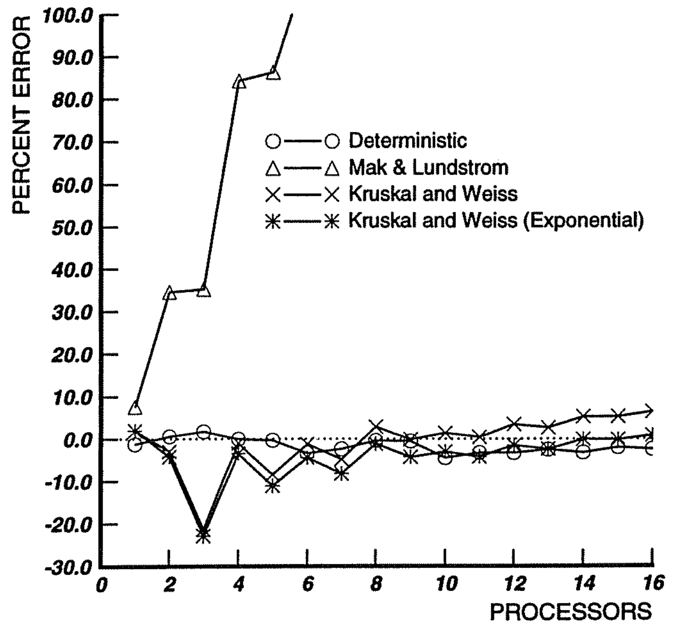
would be even higher then with the condensed graph because of the scheduling assumptions in the model, as explained in Section 5.1.3.)

The condensed graph was also used as input to the deterministic model, and the results obtained were identical to those from the original task graph with the same model. This corroborates our discussion in Section 4.2.2, where we had predicted that these results would be identical when identical resource usage parameters are used for each set of tasks condensed into a single node in the condensed graph, as was done in our experiments with this application.

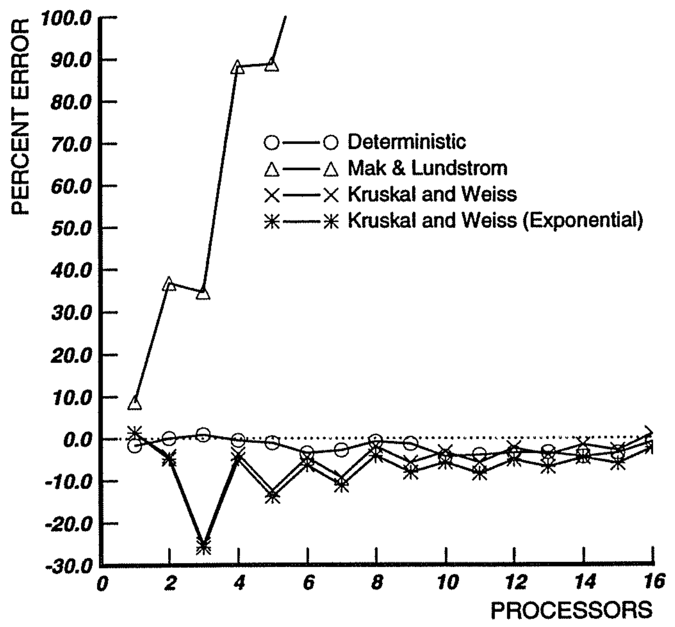
5.2.2. Results for PSIM

We consider two input network sizes for PSIM, corresponding to 1024 and 4096 processors respectively. The task graph contains 10243 and 40963 tasks in these two cases, with almost exactly half in each phase. The percentage errors in the predictions for the two input sizes are shown in Figure 5.3 (a) and (b).

The KW_{actual} model is less consistently accurate for this program than for MP3D. For small odd values of P, this model (as well as KW_{exp}) underestimate the execution time because the model scheduling assumptions do not capture the unequal amounts of work allocated to the even and odd numbered processors by processor-splitting. At larger values of P, this error becomes small, and the model is accurate for these input sizes. In fact, the model does not represent the scheduling precisely even at larger values of P because the scheduling in the model is oblivious of the *specific* task times, whereas in the actual program, the loops with large task granularities are evenly divided between two sets of processors to minimize load imbalance. The large common variance used to represent the highly skewed loop granularities, combined with the oblivious task scheduling, implies that the model could *potentially* predict higher load balance than in the actual program if the number of tasks executed per process per phase were small. For the input sizes and number of processors considered here, this number is very large and hence this source of error is not significant, although it introduces slightly higher error in the smaller input size.



(a) Input Size: 1024 nodes



(b) Input Size: 4096 nodes

Figure 5.3. Errors in the Predicted Running Times for Program PSIM.

The KW_{exp} model has the same potential source of error at the larger values of P, but these are actually smaller than in the KW_{actual} model because the variance of the exponential is even smaller than the variance used to represent the highly skewed task times in KW_{actual}. Thus, in fact, the number of tasks per process per phase is large enough to make the exponential task assumption fairly accurate for these inputs sizes. As with KW, however, the accuracy may not be consistent across different input sizes and different numbers of processors, for the same program.

The accuracy of the three Markov chain models cannot be directly inferred from KW_{exp} for this program (as it could for MP3D). Qualitatively, however, we expect those models to be as accurate as the KW_{exp} model because the effect of the exponential task assumption is relatively small, due to the large number of tasks per phase. However, the large task graphs of PSIM would clearly preclude applying the Markov chain models directly to the original graph.

The condensed graph for this application (on 16 processors) has 195 tasks. (When condensing the graph, we accurately represent the processor-splitting scheduling method used in the program.) However, the results of the *ML* model show that when the condensed graph is used, the pessimistic assumption of exponential tasks is as severe for PSIM as it is for MP3D. As before, these results show that using the condensed graph is not a viable option in exponential task models.

In contrast to the above models, the deterministic model is *consistently* accurate, yielding execution time predictions within 3 to 4% of the actual measured values. Thus, even the highly skewed task granularities of PSIM are accurately represented (along with mean communication costs) by a set of deterministic values. The model also accurately represents the task scheduling method. Ignoring the variance due to communication delays again appears reasonable, even though this application has relatively high communication overhead due to bus contention (e.g., bus utilization is 0.81 for the larger input size on 16 processors).

The condensed graph was also used as input to the deterministic model and, as with MP3D, the results obtained were identical to those from the original task graph.

5.2.3. Results for Locus Route

As seen in Section 3.3, non-deterministic CPU requirements can cause the execution time of Locus Route to vary significantly from one execution to the next, for the same input. This is illustrated in Figure 5.4, which gives a histogram of the measured execution times in 150 runs of Locus

Route on 16 processors, for the input circuit bnrE.grin. For such a program, comparing model predictions against a single measured metric such as the mean alone will not be sufficient to provide a complete picture of model accuracy. Thus, we compare model predictions against the *range* of measured speedups, i.e., the minimum and maximum, as well as the mean. (For these comparisons we use the absolute values of speedup, rather than percentage errors.) To compare the speedup predictions from different models, all speedup values are computed relative to the *mean measured* execution time on 1 processor.[18] 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 5.5 compares the speedups predicted by the *KW*, KW_{exp} and Deterministic models with the range and the mean of the measured speedups, The *ML* model could not be used because the condensed task graph cannot be constructed for any of the task scheduling options implemented in the program, since they are all dynamic scheduling disciplines. Before discussing the accuracy results, note from the figure that the difference between maximum and minimum speedup is fairly small for most values of *P*, and furthermore the mean is close to the maximum in all cases, showing that most measured values are clustered close to the maximum speedup or minimum execution time (just as in the histogram of Figure 5.4). Thus, comparisons against the mean would also be meaningful, at least for this input, although the more complete picture provided by Figure 5.5 should not be ignored.

The KW_{actual} model shows higher errors here than for the two previous applications. In this application, skewed task times combine with the *ordering* of tasks to introduce a greater load imbalance in the actual program than is predicted by the model using a common variance alone. 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. Since the model assumes i.i.d. tasks with a common mean and variance to represent skewed task times, its scheduling assumptions cannot distinguish different orderings of the same set of tasks (even though the FIFO scheduling algorithm in the program is exactly equivalent to scheduling in the model with

18. The comparison is made in terms of speedups rather than execution time because the minimum, maximum and mean execution time curves are difficult to distinguish at high values of *P*, where the execution times fall near the low end of the Y-axis scale.

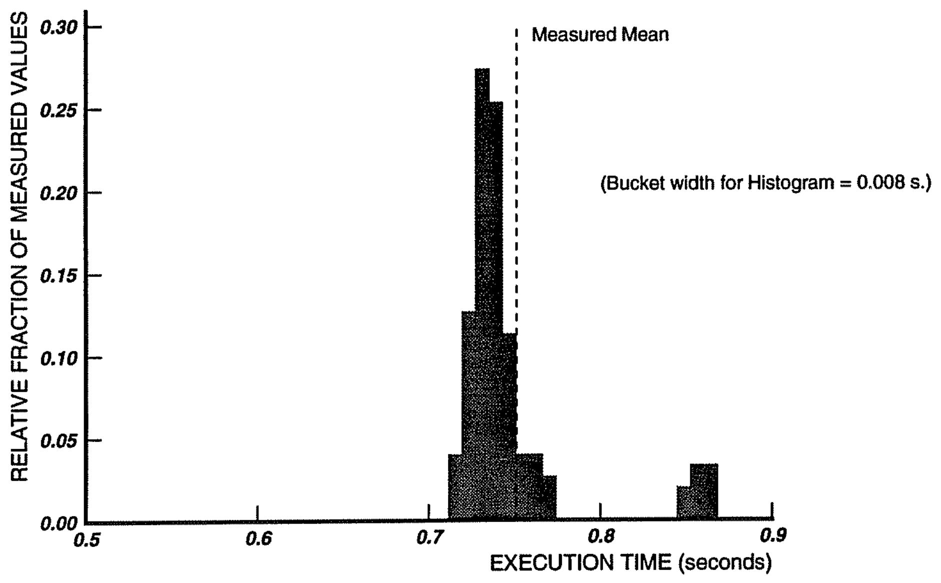


Figure 5.4. Histogram of Execution Times in Different Runs of Locus Route.

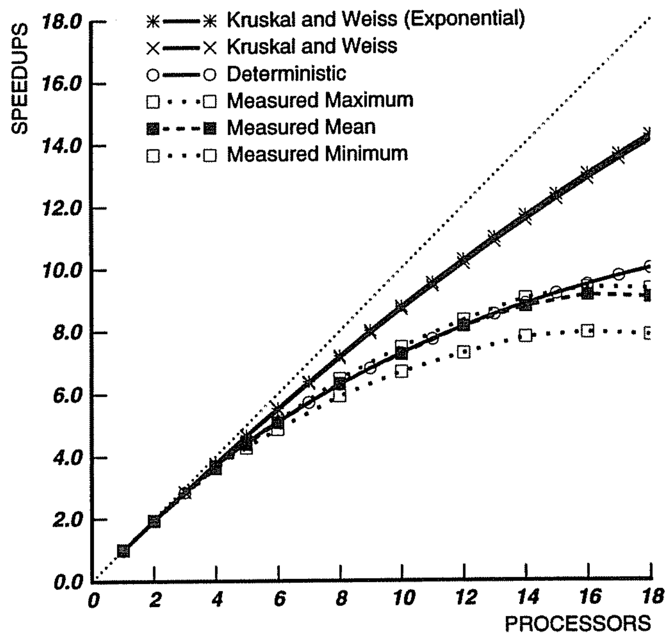


Figure 5.5. Comparison of Predicted and Measured Speedups for Locus Route.

Input circuit: bnrE.grin

a batch size of 1). Thus, the mean execution time predicted by the model is significantly smaller than observed in practice for this particular ordering of tasks. Correspondingly, the predicted speedups in Figure 5.5 are significantly higher than even the maximum measured values.

The error in the KW_{exp} model arises for the same reason, and predicts speedups almost identical to the actual variance model. Thus, in this application, the exponential assumption correctly represents the total variance due to skewed task CPU times as well as communication delays. However, the exponential assumption will have too high a variance for representing variations due to communication delays alone. Thus the three Markov chain models would over-estimate the execution time (or underestimate the speedup) of this program, and furthermore this error would be significant since this program has even fewer tasks per process per phase than MP3D.

Finally, the Deterministic model is quite accurate for this program compared to the mean measured values, but this error shows an increasing trend at higher values of P. In addition, the error compared to the most distant measured value (the minimum speedup or maximum execution time) is somewhat higher than in the two previous programs, due to the variability in task CPU requirements observed in Section 3.3. Further detailed measurements in individual executions indicated that the increasing trend in the error at higher values of P is also attributable to the variability in CPU requirements (which arises because the task CPU requirements are sensitive to the order of execution of tasks). For example, we measured the individual task execution times in a specific execution on 16 processors and subtracted the *measured* communication costs (available from the detailed hardware measurements used in Chapter 3). The remaining values, which correspond solely to CPU usage, were significantly higher than the measured CPU requirements on 1 processor. Since the latter are used as inputs to the model, the predicted execution times are low. Similar measurements on 2, 8 and 18 processors showed that this effect is most significant at P=16 or P=18 but much less significant at P=2 or P=8, matching the observed trend in the model prediction errors.

Even for such a program, however, we believe that at present the deterministic model is still the model of choice, because of the inaccuracy and/or inefficiency of current stochastic models. Even if, in the future, an accurate and efficient stochastic model were available that provided the capability to represent the variability in each task's CPU requirements, quantifying the variability in individual task times due to factors such as sensitivity to the order of execution appears difficult. Thus in practice, a stochastic model may not prove more useful than a deterministic model that ignores this variability.

In general, if the deterministic model is being considered for use with another program having significant variability in CPU requirements, it will be necessary to informally understand or formally quantify the variability in execution times of the program so as to understand how the model predictions compare with the actual range of execution times, before being able to draw broader conclusions about program performance from the model predictions. For Locus Route in particular, the model predictions are sufficiently accurate (both qualitatively and quantitatively) for the model to be potentially useful for exploring program performance issues. In fact, we use the model in Section 7.3 to evaluate various interesting changes to this program.

5.2.4. Comments on a Previous Deterministic Model

Our comparisons above have focused on contrasting previous stochastic models with the deterministic model developed in Section 4. We have not directly compared the most relevant previous deterministic model, namely that of Tsuei and Vernon [TsV90]. The main reason is that their model has already been shown to be accurate for fork-join programs with good load-balance, and it is restricted to these programs. More specifically, as explained in Section 2.4, the model assumes that the computational work as well as communication costs and other overheads are equally divided among the processes in each parallel loop. Therefore, although it could be applied to other fork-join programs, it would not be accurate for programs that have significant load imbalances.

In fact, we can easily predict the accuracy of their model if it *was* applied to the three fork-join programs studied above. The model would be fairly accurate for MP3D, but still slightly more optimistic than KW_{actual} or the Deterministic model, because the load-imbalances in this program are small yet measurable. For PSIM, the model would require a slight extension to apply. Specifically, to be able to accurately model the processor-splitting scheduling, the calculation of CPU time per process per phase would have to be carried out separately for the even and odd numbered processes in each phase. This extension would allow the model to capture the load-imbalance due to processor-splitting fairly accurately, even at odd values of P. Thus, this extended model should be fairly accurate for all values of P. Finally, for Locus Route, the model would significantly underestimate the execution time because it cannot represent the load-imbalance. In particular, it would have larger (more negative) errors than the KW_{actual} model because both models use the same value of the common mean task time, but the KW_{actual} model also partially represents the skew in task times using the common variance.

5.2.5. Comments on Previous Stochastic Bounding Techniques

We have also not directly compared the two previous stochastic bounding techniques discussed in Section 2.6. Recall that these techniques are applicable when the maximum parallelism in the input graph is no larger than the number of processors. Thus, these techniques could be applied using the condensed graph for MP3D and PSIM as well as for DynProg, since all three applications use static scheduling of tasks, but not for Locus Route and Polyroots.

These bounding techniques share some limitations observed for the stochastic models studied above since they require the same or more detailed inputs as the above models. In particular, they require the full task graph along with distribution of the individual task times. On the one hand, when accurate (i.e., possibly unequal) individual mean task times are used, the variance of task times only represents variations due to communication and contention delays and, in some programs, due to individual task CPU requirements as well. However, the results for the deterministic model above, as well as the study of random delays in Chapter 3, show that the former source of variance can be ignored without significant loss of accuracy. In any case, measuring or estimating the variance due to either source is difficult in practice. On the other hand, when a *common* task time distribution is used, as in the Kruskal and Weiss model, the results for that model show that the accuracy of the predictions will be inconsistent across different programs as well as across different numbers of processors and input sizes for specific programs. In addition to the above limitations common to stochastic models, the data on the accuracy of these specific bounds previously provided by the authors show that the tightness of the bounds can decrease with the parallelism in the input graph and is sensitive to the specific task graph structure.

The deterministic model uses little or no more detailed information than used by these bounding techniques (even when the bounds are applied using a common task time distribution, as mentioned above). Furthermore, the deterministic model appears consistently accurate (as our results for the two other programs will further confirm), and is extremely efficient. Finally, it is not restricted to statically scheduled programs. Thus, in cases where the conclusions of the random delays study holds, the deterministic model improves significantly on the bounding techniques in many of these respects. If an application domain arises where significant variance of individual task execution times is possible, a stochastic model may be necessary and if the requirement of a condensed graph can be satisfied, the bounding techniques would be a promising approach. However, further data would still be needed to evaluate the accuracy of these techniques for actual programs.

5.3. Evaluation of the Deterministic Model for Programs with Complex Task-Graphs

The two remaining programs, Polyroots and DynProg, have non-series-parallel task graphs as explained in Section 5.1.1. The only previous general analytical models that apply to such programs are the three Markov chain models. (Although the model of Lewandowski et al. [LCB92] does apply to DynProg, it was specifically developed for this program and is restricted to similar pipelined programs. Hence we do not consider it here.) The former three models are too complex to be used for these programs. Specifically, the original task graphs in these programs (ranging in size from 217 to about 25000 tasks) are too large to apply these three models directly, as the discussion in Section 5.2 indicated.[19] Furthermore, a condensed graph is not possible for Polyroots since it is dynamically scheduled, and the condensed graph for DynProg is almost the same as the original graph because of the pipelined precedence constraints. Thus, it does not appear practical to use previous stochastic models to analyze these two programs.

On the other hand, the deterministic model can be used quite easily for these programs. In fact, as explained earlier, Polyroots and DynProg have relatively low communication costs and other overheads on the Sequent Symmetry, and we can even apply the basic deterministic model. For the program Polyroots, having task graphs with 217 tasks and 348 tasks for two input sizes, solving the basic deterministic model for all values of P from 1 through 16 is virtually instantaneous. The program DynProg has much larger task graphs: 1403 and 32003 tasks for two input sizes studied. Even in the latter case, the basic model can be solved in about 30 seconds of execution time on a DECstation 5000/125. In fact, this case required the longest solution time and the largest memory capacity (about 6 megabytes) of all the deterministic model analyses presented in this thesis. (Though PSIM has larger task graphs up to about 40000 tasks, its graphs are significantly simpler, containing large groups of tasks with similar behavior that can be manipulated much more efficiently, as explained in Section 4.3.)

19. For example, just the 20 tasks in the final phase of Polyroots for the smaller input size would induce 2^{20} states in the Markov chain at $P = 20$, and this portion of the state space cannot easily be collapsed because the tasks in the phase have widely varying CPU requirements.

5.3.1. Results for Polyroots

The accuracy of the basic deterministic model for program Polyroots is shown in Table 5.2. The table gives the measured and predicted execution times, as well as the percentage error in the predictions, for each of two input sizes, namely input polynomials of degrees 20 and 30. 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 the program has some small forking and communication overheads and these are relatively more significant with the smaller input size and greater parallelism. Overall, the results are extremely accurate and show that the basic deterministic model precisely represents the task execution sequence of the program, i.e., the task execution times and dynamic task scheduling.

Using this program, we conducted an additional experiment to evaluate how accurately the full deterministic model represents contention for a shared software resource, namely the lock protecting the shared task queue in Polyroots. An important difference from the bus and memory contention modeled in the previous sub-section is that very few lock accesses occur between synchronization points; thus the renewal model in Chapter 3 shows that the relative variance introduced by lock accesses will be larger than that introduced by the numerous remote memory accesses. To ensure that the cost of each lock access had significant mean and variance, we artificially introduced an exponentially distributed delay into the lock holding time (the actual contention for this lock in the program is small). The mean delay time can be specified as an argument to the program. No new measurements of the model inputs were required because the tasks were the same, and the lock holding times were known.

We modeled the lock as an M/M/1/∞/K queue, which forms the system-level model. We maintained a running average of the number of active processors in the execution sequence and used this as the value of K to solve the queueing model at each step. (This contrasts with using the instantaneous number of active processors for the Sequent bus queueing model in the previous sub-section, and is perhaps better since the lock is accessed only at task boundaries.) At each step, the mean lock response time was included in the mean task execution time of each task added to the set E_{set}.

The predicted and measured running times on 2 through 16 processors are shown for the two input sizes in Table 5.3 (a) and (b) respectively. The mean lock holding times used were 1, 10 or 50 milliseconds; the latter value far exceeds the execution time of many of the tasks in the program.

Table 5.2. Accuracy of the basic deterministic model for program Polyroots.

Para-meters	Procs	Measured Time (s)	Predicted Time (s)	Error (%)
Degree = 20	1	99.278	99.158	-0.12 %
	2	51.384	51.033	-0.68 %
	4	27.558	27.214	-1.25 %
217 tasks	8	16.705	16.579	-0.75 %
	12	11.850	11.55	-2.5 %
	16	11.558	11.01	-4.8 %
Degree = 30	1	397.95	397.62	-0.08 %
	2	202.51	201.66	-0.42 %
	4	107.46	106.76	-0.65 %
348 tasks	8	56.44	56.21	-0.41 %
	12	44.38	44.08	-0.67 %
	16	31.69	31.84	+0.47 %

Table 5.3. Accuracy of deterministic model for program Polyroots with lock contention.

(a) Input Polynomial Degree 20

Lock Delay	Procs	Measured Time (s)	Predicted Time (s)	Error (%)
1 ms.	2	51.52	51.15	0.73 %
	4	27.57	27.28	1.04 %
	8	16.78	16.36	2.48 %
	12	11.92	11.55	3.09 %
	16	11.60	11.03	4.84 %
10 ms.	2	52.67	52.55	0.23 %
	4	28.57	28.27	1.05 %
	8	17.91	17.40	2.82 %
	12	13.23	12.58	4.97 %
	16	13.14	12.22	7.00 %
50 ms.	2	59.36	59.41	-0.09 %
	4	35.14	34.99	0.43 %
	8	25.53	24.96	2.24 %
	12	22.22	20.57	7.43 %
	16	22.58	20.03	11.29 %

(b) Input Polynomial Degree 30

Lock Delay	Procs	Measured Time (s)	Predicted Time (s)	Error (%)
1 ms.	2	202.75	201.84	0.45 %
	4	107.55	106.86	0.64 %
	8	56.51	56.19	0.57 %
	12	44.45	44.12	0.73 %
	16	31.61	31.86	-0.80 %
10 ms.	2	204.58	203.58	0.49 %
	4	108.64	107.92	0.67 %
	8	57.40	57.28	0.22 %
	12	45.79	44.86	2.04 %
	16	33.62	32.45	3.47 %
50 ms.	2	213.76	212.32	0.68 %
	4	115.39	116.16	-0.67 %
	8	65.71	65.30	0.63 %
	12	54.92	53.98	1.70 %
	16	44.45	42.82	3.68 %

Nevertheless, the model errors are again quite small, and are higher than 5% only when lock holding times are large (10 ms. or more) and total processing time is small (degree 20). Note that the total running time of the program is substantially higher than in Table 5.2, indicating that there was significant contention for the lock.

Table 5.4. Accuracy of the basic deterministic model for program DynProg.

Parameters	Procs	Measured Time (sec)	Predicted Time (sec)	Error (%)
Sequence	1	0.457	0.465	1.817 %
Length	2	0.237	0.236	-0.712 %
$G = 100$	4	0.122	0.122	-0.0755 %
	8	0.066	0.066	0.723 %
1403	12	0.0482	0.0472	-2.231 %
tasks	16	0.0477	0.0453	-2.908 %
	18	0.0464	0.0451	-2.758 %
Sequence	1	11.503	11.679	1.526 %
Length	2	5.856	5.852	-0.0803 %
$G = 500$	4	2.939	2.941	0.0905 %
	8	1.496	1.499	0.193 %
32003	12	1.008	1.017	0.849 %
tasks	16	0.773	0.785	1.615 %
	18	0.686	0.7003	2.068 %

5.3.2. Results for DynProg

For the second program, DynProg, only the accuracy of the basic deterministic model was evaluated, using two input sizes corresponding to sequence lengths of 100 and 500. The predicted and measured execution times, as well as the percentage errors in the predictions are listed in Table 5.4. 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.

Overall, the results for these two programs show that the deterministic model can be used to efficiently and accurately solve large and complex task graphs that are impractical to analyze using previous stochastic models.

5.4. Summary of the Results

We have applied the Deterministic model as well as representative stochastic models to three parallel programs, to compare the generality, accuracy and efficiency of the various models. We also applied the deterministic model to two further programs that are impractical to analyze using previous stochastic models. Our key conclusions are as follows:

1. Simple stochastic models based on i.i.d. tasks and incorporating the actual variance of task times (such as the Kruskal and Weiss model) are extremely efficient, and are also accurate for programs that do not exhibit large skew in task times and with task scheduling that matches the model assumptions. However, significant errors can be introduced for programs where specific details of task skew, task ordering and the task scheduling algorithm cannot be represented by the simplified assumptions in the model. Furthermore, all such models are restricted to programs with the simplest task structures [AIA90, KrW85, MaS91].
2. Stochastic models applying to more general classes of programs [KME89, MaL90, Moh84, ThB86], all of which assume exponentially distributed task times, have significant and sometimes large errors. Furthermore, the models in this class appear too inefficient to use even for programs with relatively small task graphs of a few hundred tasks. Applying such models using a condensed task graph (when this can be constructed) does not appear useful because the exponential assumption produces extremely high errors in this case.
3. The Deterministic model is accurate in all the programs studied here, because it accurately represents key details of task scheduling, non-uniform task times, and average communication costs. Ignoring the variance in task times due to communication and contention delays has little or no perceptible impact on the model accuracy. The model is very efficient, and is easily able to solve even task graphs with tens of thousands of tasks. Finally, it is applicable to a wide class of programs, including programs that appear impractical to analyze using the stochastic models.

The above results provide evidence for answering a more general and fundamental question: *is a deterministic approach likely to prove more practical and widely applicable than stochastic approaches for parallel program performance prediction*? Our results show that all previous stochastic models except those that are restricted to fork-join programs with simple task-scheduling are inefficient, inaccurate, rather ad hoc and conceptually complex. In all cases, the complexity is inherently produced by the stochastic representation. In contrast, the Deterministic model appears to have to be accurate and efficient for a wide variety of programs, and it is conceptually simple to implement and use as well. Again, the conceptual and practical simplicity is inherently due to the deterministic task assumption.

Chapter 6

Comparison of Deterministic Model and Parametric Bounds

Techniques to obtain bounds on the speedup of a parallel program, as described in Chapter 2, provide another, somewhat different, modeling approach that can also allow a programmer to gain some insight into program performance. An important example of such techniques is the work of Eager, Zahorjan and Lazowska [EZL89], described in Section 2.6. In general, these techniques compute upper and lower bounds on speedup from simple program metrics such as average and maximum parallelism. Thus, these techniques are somewhat different in nature from the detailed analytical models evaluated in the previous chapter: they are intended to provide general qualitative insights into program performance rather than detailed quantitative measures, and they are based on a few abstract parameters describing an application rather than a detailed description such as a task graph. Therefore, the two questions we address in this comparison are: Are these bounding techniques fundamentally easier to use than a detailed model, when evaluating a parallel program? And how much more qualitative and quantitative information is provided by a detailed model, compared to the bounding techniques?

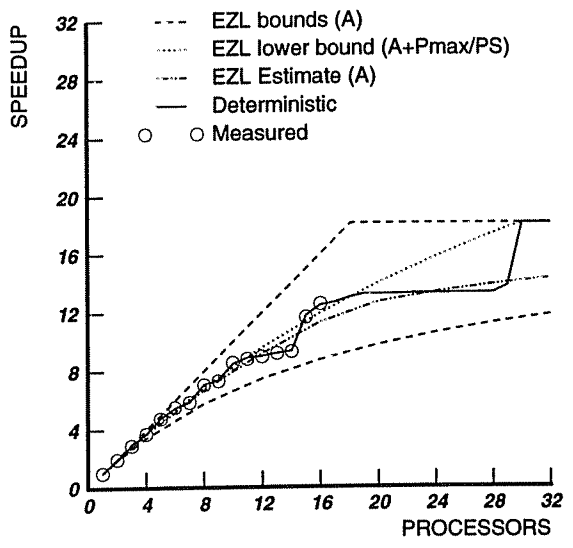
When applying the EZL bounds to a given task graph, the key parameter required is the average parallelism, A. Using the definition given, A can be computed as the ratio of the execution time on 1 processor to the execution time on $P \geq P_{\max}$ processors (where $P_{\max}$ denotes the maximum parallelism). However, directly measuring the latter quantity is not possible for typical scientific and engineering applications because $P_{\max}$ at realistic input sizes is often very high (e.g., in the thousands or more). On the other hand, computing these metrics for a particular program is *no simpler than a single solution of the basic deterministic model* for that program. The information required to compute these metrics is exactly what is required to solve the basic deterministic model, namely knowledge of the task graph structure and the mean execution time of the tasks. (For example, consider estimating A for the task graph of Polyroots in Figure 5.1(d).) Given this information, the basic model of Figure 4.1 provides a simple algorithm for computing average parallelism (the speedup at $P = P_{\max}$) and more detailed metrics as well. Thus, obtaining the required metrics for the bounds is equivalent to a single solution of the deterministic model.[20] Furthermore, the deterministic model is efficient enough that multiple such solutions for different values of P is quite straightforward, even for fairly large task graphs. Thus, it is not significantly more difficult to obtain the required model inputs and solve the deterministic model multiple times (i.e. obtain precise speedup estimates from the model) than to obtain the same inputs and solve the model once (in order to compute the bounds). We conclude that comparable effort is required to apply the two approaches in practice.

In the remainder of this chapter, we use examples to compare the usefulness of the two approaches for understanding program performance. Our first example is the program Polyroots, which is particularly interesting for a number of reasons. First, $P_{\max}$ is small (e.g., 30 for the larger input set used earlier), making it possible to compare the two approaches over the full range $1 \le P \le P_{\max}$. Second, the program fits well with both assumptions used to derive the bounds denoted $\langle A \rangle$ in Section 2.6. Specifically, overhead costs are very low in this program (as shown by the accuracy of the *basic* deterministic model for this program in Table 5.2), and the dynamic scheduling of tasks used in Polyroots is work-conserving. However, the processor sharing assumption for the $\langle A + P_{\max} | \text{PS} \rangle$ bound does not hold.

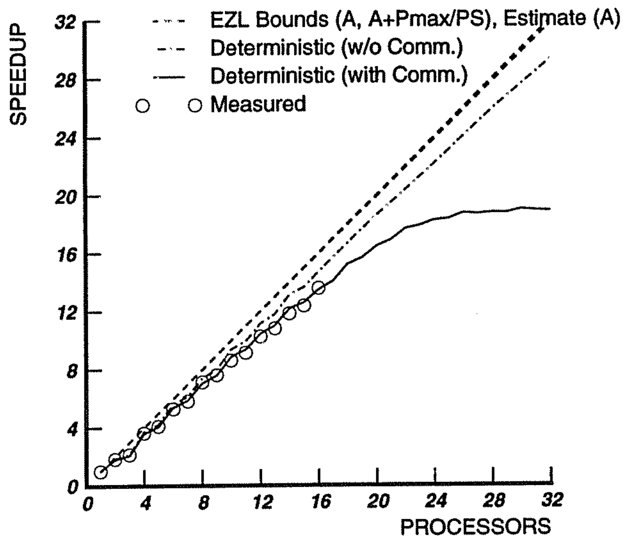
To compute the bounds, we derived the required parameter values exactly as described above. Specifically, we used the basic deterministic model to predict the execution time, T_∞, of Polyroots on $P = P_{\max} = 30$ processors ($P_{\max}$ is usually easy to derive by inspecting the task graph. Alternatively, the deterministic model can be solved using $P = N$, which gives T_∞ as well as $P_{\max}$.) The execution time on one processor is just the total processing requirement, $T(1) = \sum_{i=1}^{i=N} T(i)$, and $A = T(1)/T_\infty$. Thus, as explained above, A can be derived from a single solution of the basic deterministic model ignoring communication.

---

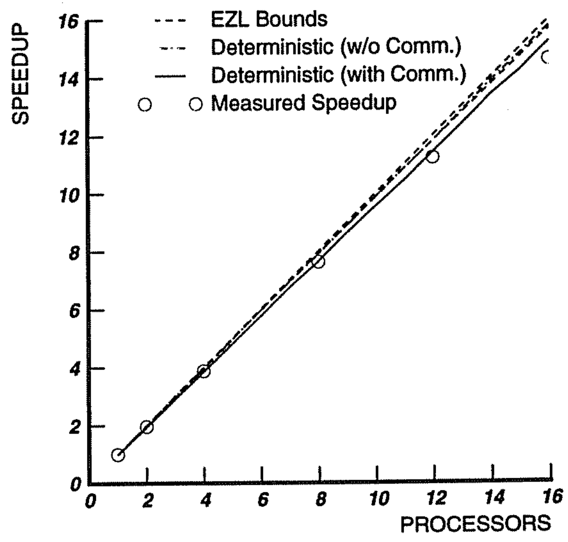
20. An interesting, related measurement-based technique has been developed by Larus and implemented in a tool called pp [Lar93]. His technique uses a trace of an execution of a *sequential* program to detect data dependencies and CPU times for the iterations of each parallel loop in the program, and computes the potential speedup of the program (if parallelized) on an idealized parallel computer with an unbounded number of processors that communicate and synchronize at no cost via a uniform-cost shared memory. Thus, his technique can be viewed as constructing a task graph by detecting the tasks and precedences within the loops in a program (with the additional constraint of preserving the sequential semantics of the program) and actually *computing the average parallelism* of the program, if it were parallelized. In fact, the computation of the average parallelism is exactly equivalent to the solution of the basic deterministic model on an unbounded number of processors, since both techniques just compute the critical path in the task graph.



(a) Polyroots
$N = 317, A = 18.1, P_{\max} = 30$



(b) PSIM
$N = 40963, A = 3407, P_{\max} = 20480$



(c) MP3D
$N = 1978, A = 726, P_{\max} = 1274$

Figure 6.1. Comparison of deterministic model and parametric speedup bounds.

In Figure 6.1 (a), we give the speedup bounds ($\langle A \rangle$ and $\langle A + P_{\max} | \text{PS} \rangle$), the speedup estimate which is the geometric mean of the $\langle A \rangle$ bounds, as well as the measured speedup and the actual speedup predicted by the deterministic model (recall that the latter is theoretically exact in the absence of communication and other overheads), for program Polyroots on the larger input size. Although the speedup estimate from the $\langle A \rangle$ bounds is quantitatively fairly accurate for this program, qualitatively, the estimate and bounds do not capture the most interesting aspect of program behavior. Specifically, the actual speedup shows significant jumps for many values of P, most noticeably from 14 to 15 processors and from 29 to 30 processors. These jumps occur because the final phase of the program has 30 large tasks and therefore the load-balance is relatively poor at values of P that are not submultiples of 30. In contrast to the bounds, the deterministic model represents this phenomenon exactly. In fact, in Chapter 7, we use these insights to improve the overall performance of the program, as well as to reduce the non-uniformity in the speedup.

We next consider the program PSIM for the larger input size (4096 nodes in the simulated network). This program has much higher values of $P_{\max}$ and A: 4096 and 3407 respectively. Furthermore, it violates both basic assumptions used to derive the bounds because it has significant bus contention overhead and it uses static scheduling of tasks which is not work-conserving. Thus, PSIM differs from Polyroots in all these respects. To apply the bounds, we estimated A by using the basic deterministic model (i.e., ignoring overheads), just as for Polyroots.[21]

In Figure 6.1 (b), we give the $\langle A \rangle$ and $\langle A + P_{\max} | \text{PS} \rangle$ bounds, the $\langle A \rangle$ estimate, the measured speedup, the speedup if communication were absent (predicted by the basic deterministic model), as well as the predicted speedup including communication. The first point of interest in the figure is that, since $P \ll A$, the $\langle A \rangle$ bounds and estimate as well as the $\langle A + P_{\max} | \text{PS} \rangle$ bound all indicate almost-linear speedups, and the only information they provide is that there is sufficient available parallelism in the program for near-perfect speedups, if the parallelism can be efficiently exploited during execution. These are fairly severe limitations because many scientific and engineering applications will have $A \gg P$ in systems of practical interest. (For example, the corresponding results for

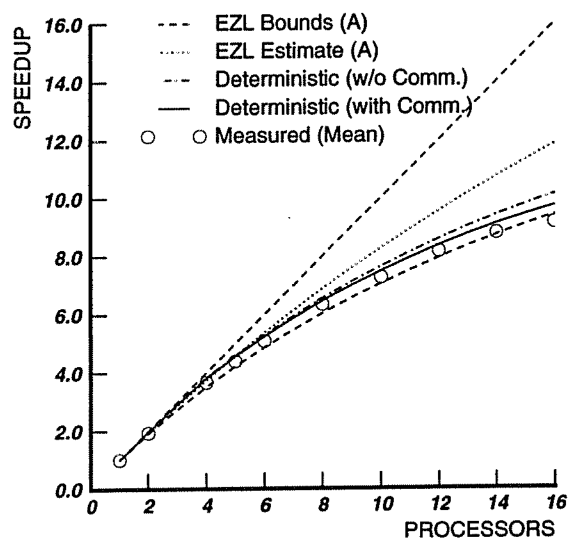
21. Note that the overhead costs cannot be included in the speedup bounds by including them when estimating A for this program (e.g., by using the full deterministic model to compute T_{∞}, instead of the basic model). This is because the communication and contention costs on $P_{\max} = 3407$ processors would be prohibitively high, even on a more scalable system of the near future. Thus, such a value of A is not likely to provide meaningful bounds.

MP3D below show the same limitations.) Quantitatively, in fact, even the speedup that would be obtained *if communication were absent* lies significantly lower than the lower bound because of the non-work-conserving scheduling of the tasks. The true speedup (including communication costs) is lower still, as shown by the deterministic model as well as the measured values for $P \leq 16$. Qualitatively, the bounds and speedup estimate cannot represent the non-uniform speedup curve caused by the higher load-imbalance at odd numbers of processors, whereas the deterministic model captures it very accurately. Another important qualitative conclusion shown by the full deterministic model, which cannot be obtained from the bounds, is that application speedup would be severely limited by bus contention overhead with a greater number of processors on this system.

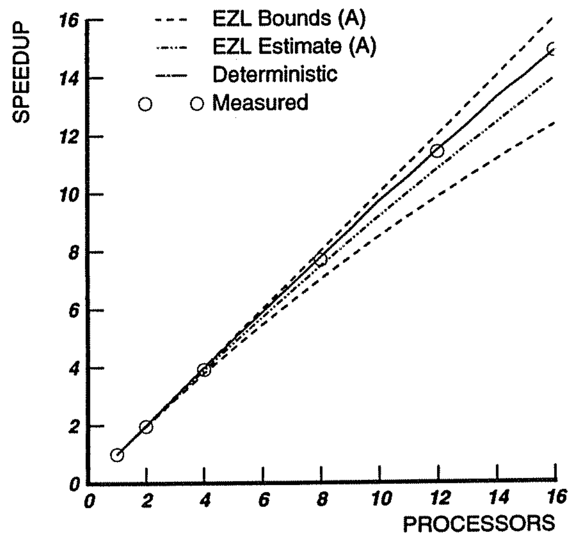
The corresponding curves for MP3D with the larger input size, given in Figure 6.1 (c), show results similar to those obtained for PSIM: the $\langle A \rangle$ bounds predict almost perfect speedup due to high available parallelism in this program, since $A = 726 \gg P$. Like PSIM, this program is also statically scheduled, but the predicted speedup ignoring communication, which falls within the bounds, shows that the static task allocation achieves good load balance for these values of P, i.e. the application is able to exploit the available parallelism well. The detailed model including communication, however, shows a small but perceptible loss in speedup due to communication costs, and this is corroborated by the measured values of speedup.

The results above show that with very similar effort to that required to calculate A, exact estimates of the speedup can be obtained using the deterministic model. The exact estimates provide important qualitative information not available from the bounds, and are quantitatively more accurate. Furthermore, the bounds do not hold for statically scheduled programs (and other non-work-conserving disciplines), programs in which contention overheads are significant, or programs that have different task graphs for different values of P, whereas the deterministic model is applicable in all these cases.

In the remaining two applications, Locus Route and DynProg, the predicted speedups from the deterministic model do not provide significantly greater insight than the bounds and estimate, although they are quantitatively more accurate. Both applications have relatively low values of A, comparable to available values of P, at least for the small input sizes considered here. The predicted and measured speedups and bounds are given in Figures 6.1 (d) and (e). In Locus Route, there is low average parallelism due to the large disparity in task times, and this is manifested in a low value of $A \ll P_{\max}$ and a correspondingly low lower-bound on speedup. The poor ordering of tasks in the



(d) Locus Route
$N = 843, A = 21.4, P_{\max} = 420$

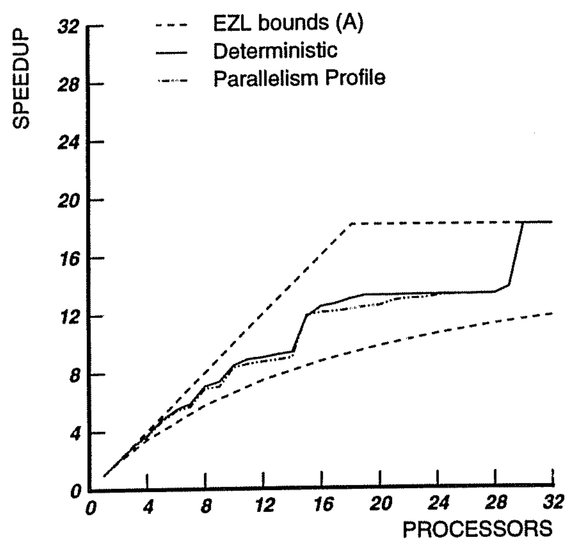


(e) DynProg
$N = 32003, A = 50.6, P_{\max} = 500$

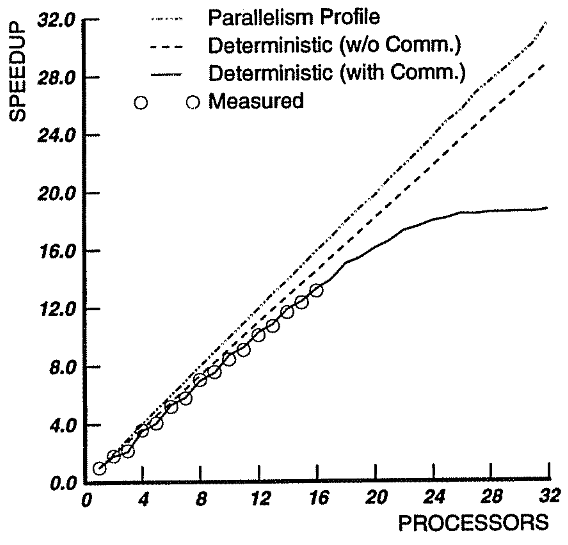
Figure 6.1. Comparison of deterministic model and parametric speedup bounds (continued).

task queue (see Section 5.2.3) causes the actual speedups to lie close to the lower-bound. (The bounds roughly hold because the dynamic scheduling is work-conserving and communication overheads are small.) In DynProg, there is somewhat low available parallelism due to the pipelined precedence constraints, which is manifested in $A \ll P_{\max}$, and a somewhat low lower bound on speedup. The predicted speedups show, however, that the static scheduling, though not work-conserving, achieves a good load-balance at these values of P, and this is confirmed by the measured speedups.

The bounds of Eager, Zahorjan and Lazowska discussed above consider only the average parallelism and sometimes the maximum parallelism. The above results show that these simple metrics are insufficient to capture many interesting details of program behavior. It is relevant therefore to ask whether bounds or estimates based on more detailed information, but still much less than the full task graph, can provide better insight. One detailed representation of the parallelism in a program is the parallelism profile, $\{f_i, 1 \le i \le P_{\max}\}$, where f_i is defined as the fraction of time that i processors are active in an execution on an unbounded number of processors. Given the parallelism profile, an estimate for the speedup may be obtained by assuming that (informally) each "level" of fixed parallelism in the graph must complete before the next level can begin. This leads to the following estimate for speedup:



(a) Polyroots
$N = 317, A = 18.1, P_{\max} = 30$



(b) PSIM
$N = 10243, A = 871, P_{\max} = 5120$

Figure 6.2. Speedup estimate based on full parallelism profile.

$$Speedup(P) \approx \frac{T(1)}{T_\infty \times \sum_{i=1}^{P_{\max}} f_i \lceil i/P \rceil} = \frac{A}{\sum_{i=1}^{P_{\max}} f_i \lceil i/P \rceil}$$

(This estimate often may be pessimistic compared to work-conserving task scheduling disciplines, but it is not a lower bound even for such disciplines.) Now, the parallelism profile is directly available from the same solution of the deterministic model used above to compute T_∞ and, therefrom, A. Thus, this more detailed estimate can be computed with essentially the same effort as the EZL bounds. To compare the information obtained from the two approaches, Figure 6.2 (a) shows the speedup estimate from the parallelism profile of Polyroots, as well as the EZL bounds and the actual speedup predicted by the deterministic model. The non-uniform speedup behavior is reflected clearly in the estimate because the leftover tasks at values of P that are not submultiples of $P_{\max}$, which lead to poor load balance in the final phase, are represented in the scheduling scenario used to derive the estimate. Thus, by using more detailed parameters, this estimate can provide more information and insight into program behavior than the bounds based on simpler parallelism parameters. A similar comparison for the program PSIM (Figure 6.2 (b)) shows, however, that the estimate is unable to capture the non-uniform speedup behavior or the loss in speedup due to load imbalance in this program, both of which are caused by the non-work-conserving task scheduling method, namely processor-splitting. The estimate obviously also cannot capture resource contention overheads. In fact, as with the EZL bounds shown in Figure 6.1 (b), the estimate provides little information when $P \ll A$. Thus, the estimate does not alleviate some of the most significant limitations of these bounds. In general, the estimate based on more detailed parallelism information *can* provide better insight into program performance in programs where the simpler bounds are applicable, but it does not significantly extend the class of programs to which these techniques apply.

To conclude this Chapter, we discuss another aspect of the relationship between the deterministic model and the work of Eager et al. and others. In their paper, Eager et al. show that the average parallelism can also be useful for making scheduling decisions in a multiprogrammed parallel system. For example, they show that allocating A processors to a program achieves a good compromise in the tradeoff between the speedup and the efficiency of the program. In a later study, Sevcik argues that in a multiprogrammed parallel system under moderate to high load, better performance is possible if scheduling decisions are based not only on A but also on one or more additional parameters such as maximum parallelism, variance of parallelism and offered system load [Sev89]. We observe that the deterministic model provides an efficient technique for obtaining these various parameters for a particular program. In fact, all these parameters can be derived from the same solution of the deterministic model used above for calculating A. Specifically, as noted above the parallelism profile of the application is derived as part of that solution, and all parameters of interest can be easily derived from this profile including the minimum, maximum, average and variance of parallelism, and the fractions of work in sequential and maximum parallel phases [Sev89]. Thus, if these scheduling results were to be used in practice, the deterministic model provides a practical means of obtaining the necessary parameters.

Chapter 7

Example Applications of the Deterministic Model

The results of the two previous chapters show that the deterministic model provides an accurate and efficient technique to predict program performance from a task-graph based description of the parallelism, scheduling and communication in the program. We argued earlier (Chapter 2) that the task-graph abstraction can be useful for exploring and understanding many program performance issues. In this Chapter, we illustrate how the task-graph based deterministic model can be used to examine several design issues for some of the programs studied in the previous chapters, specifically Polyroots, PSIM and Locus Route. For each of these programs, insight obtained by using the model led to one or more suggested program design changes. After implementing the changes for two of the programs, the performance improvement predicted by the model was borne out in each case.

7.1. Evaluating Possible Changes to Polyroots

The speedup curves for Polyroots in Figure 6.1 (a) showed that the speedup of this application is substantially less than linear, and is also non-uniform and can be particularly poor at some values of P. To gain some insight into the program in a specific instance, Figure 7.1 shows the execution profile of the program (i.e., the instantaneous number of active processors as a function of time during an execution) when executing on 24 processors. This profile is directly available from the deterministic model solution. In the figure, the fluctuating parallelism seen in initial phases of the execution is due to the small tasks and numerous precedences in the corresponding portions of the task graph. (The task graph for this input size is similar in structure to that shown in Figure 5.1 (d).) The final phase of program execution begins about 10.3 seconds after the start of the program. This phase, corresponding to the lowermost box in Figure 5.1(d), is a single parallel loop beginning and ending in a barrier, and containing $P_{\max}$ large and non-uniform tasks. The execution profile shows that there is sufficient parallelism in the initial half of the execution of this phase but much lower parallelism in the entire second half, thus indicating poor load-balance during the execution of the phase.

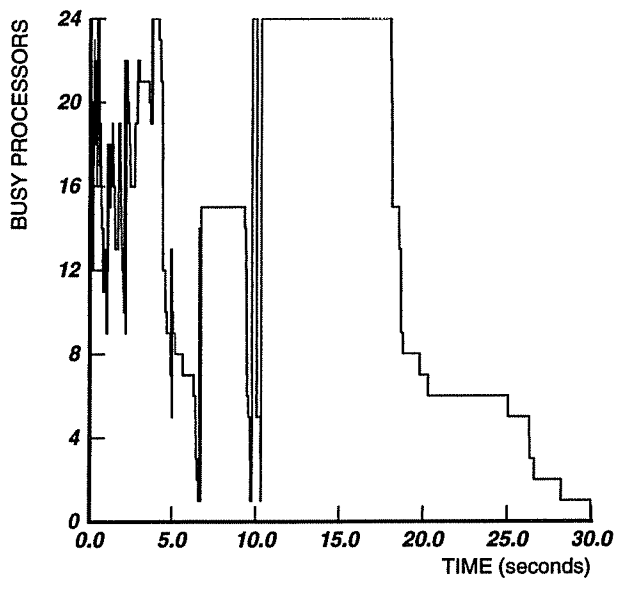
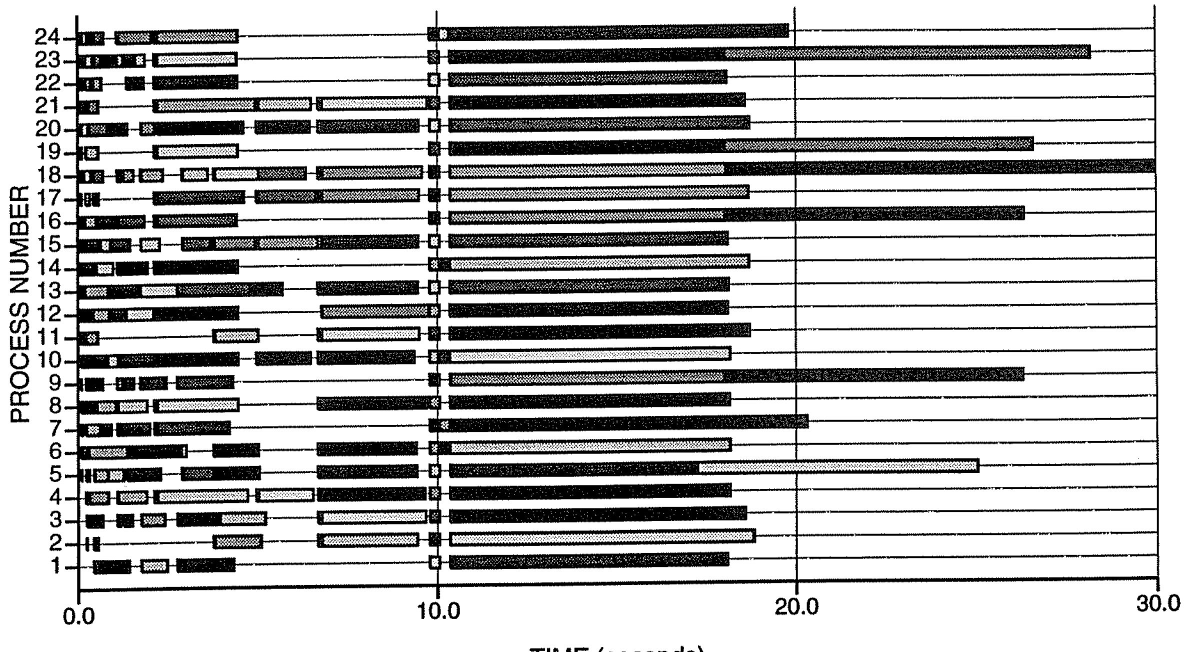


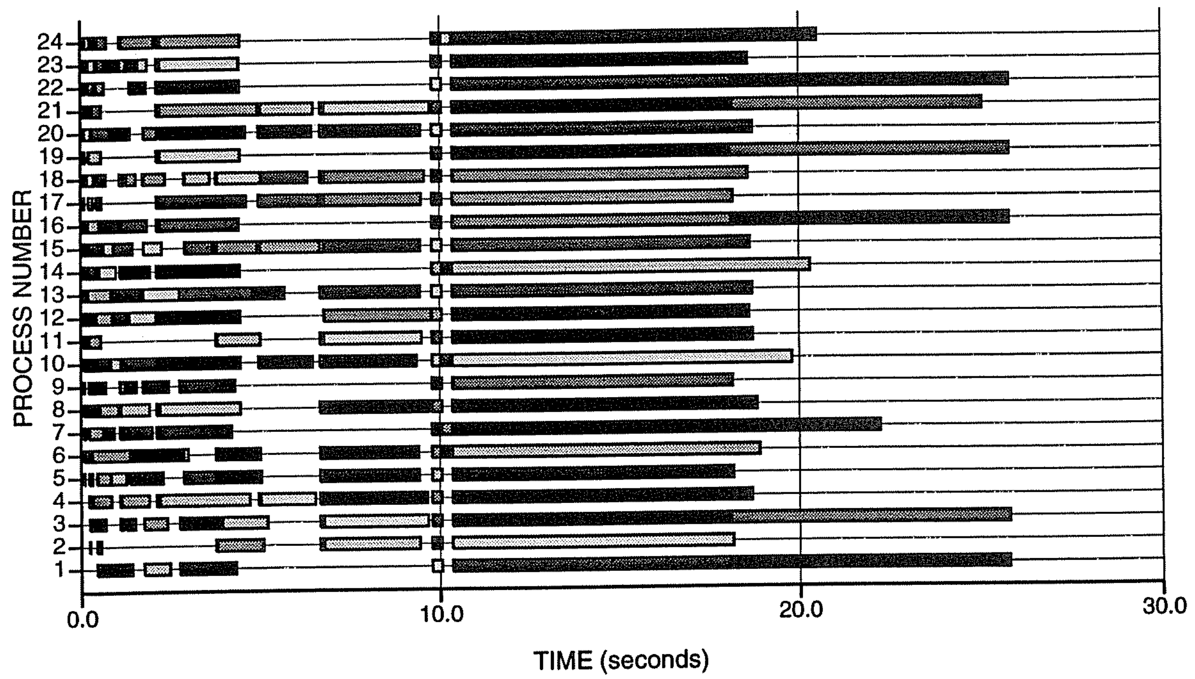
Figure 7.1. Execution profile showing number of busy processors over time for Polyroots.

Input polynomial degree 30: $N = 317$, $P_{\max} = 30$, $P = 24$.

To explore this imbalance further (and to illustrate the detailed task- and process-level information available from the model), Figure 7.2 (a) shows timelines of execution for each process; a continuous band of a fixed shade represents the execution of a single task by the corresponding process. The figure shows that the load imbalance arises because of the large, irregular granularity and relatively small parallelism. A closer inspection also shows that the two longest tasks in the phase (the last task each for processes 23 and 18) are among the last to begin execution, thus exacerbating the load-imbalance. Recall that the program uses a single shared task queue to store and allocate the tasks; the timelines show that the two largest tasks in fact appear towards the end of the queue. This immediately suggests that one simple improvement would be to place the largest task, which is trivial to identify, at the head of the task queue. However, if all task processing times in the phase were somehow known during program execution, better performance might be possible if tasks were picked off the task-queue in decreasing order of execution time. In fact, this heuristic, called the Longest Processing Time (or LPT) rule, is optimal (among non-preemptive schedules) when there are at most twice as many tasks as processors, and in the general case its execution time is bounded to within 1⅓ of that of the optimal schedule among non-preemptive schedules [HoS84].



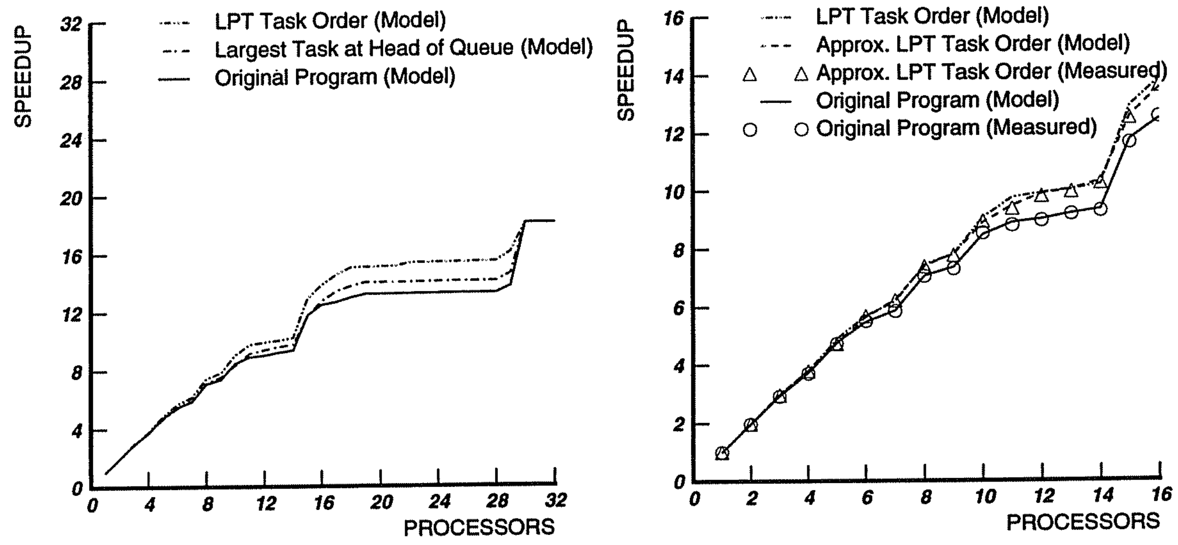
(a) Original ordering of tasks



(b) LPT ordering of tasks in final phase

Figure 7.2. Process timelines showing individual task executions for Polyroots.

Input polynomial degree 30: $N = 317$, $P_{\max} = 30$, $P = 24$.



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

Figure 7.3. Effect of reordering tasks in the final phase of program polyroots.

It should be clear that the deterministic model can easily be used to *predict* the effect of these two changes, and compare them to the original program. Observe that all three use dynamic scheduling from a single task queue, and only the ordering of tasks in the queue is different in the three alternatives. Thus, we can use the model to predict the execution time in each case simply by specifying the input values of $T(i)$ in the appropriate order. First, to re-examine the case studied earlier, Figure 7.2 (b) shows the new process timelines with the LPT rule, as predicted by the model. The figure shows that the overall execution time drops from 30 to about 25 seconds. Thus, the improvement is significant, at least for $P = 24$. To determine the effect on execution times at other values of P, the predicted speedups with the LPT order are shown in Figure 7.3 (a). The figure also shows the predicted speedup with the simpler heuristic of placing the largest of the tasks first in the queue. This graph shows that the simpler heuristic would yield a small though useful improvement for $P \geq 16$, but executing the tasks in the LPT order would yield a more significant improvement in speedup over a wide range of P. These results indicate 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 final phase executes a binary search on an interval of the real line; the two largest tasks correspond to the unbounded intervals at either end. We inserted sequential code at the start of the phase to sort the tasks in decreasing order of the corresponding interval lengths. The new task order obtained is not exactly in decreasing order of execution time; we therefore call it the Approximate LPT order. We measured the execution time of this modified program for $1 \le P \le 16$ on the Sequent Symmetry. We also used the deterministic model to predict the execution time with the approximate LPT order (simply by ordering the input data to the model in the appropriate sequence, determined from the modified program). In Figure 7.3 (b), we compare the predicted speedup for LPT with the predicted and measured speedups for the Approximate LPT task order as well as the original program. (Note that we have truncated the axes to $P \le 16$, compared to the graph in (a).) The figure shows that the simple approximation to LPT was able to realize almost the full improvement possible with LPT. More important in the 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 could be worthwhile to attempt the full task reordering. This is possible because of the precision with which the deterministic model represents task ordering and scheduling behavior, and the accuracy with which it computes synchronization costs in the program.

Improving the load balance in this phase could also be possible with a finer granularity of parallelization, but this would require modifications to the algorithm. Alternatively, this might also be possible with preemptive scheduling of tasks. For example, potentially higher and also *more uniform* speedup would be obtained if the processing power is allocated to the tasks using processor sharing (i.e., idealized round-robin task scheduling). In this case, for any number of processors, the execution time would be independent of the order in which tasks appear in the task queue since, effectively, all parallel tasks would be served "simultaneously". With system support this could be approximated by using $P_{\max}$ processes to execute the program, and system-level round-robin scheduling to schedule the processes on $P \le P_{\max}$ processors. Again, we can use the deterministic model to predict the speedup achievable with this method, using the simple modifications to the model for processor sharing described at the end of Section 4.2. These model predictions indicate the available potential for improvement; in practice, additional overheads due to context switching and cache interference between the multiple processes sharing a processor would reduce the actual performance improvement obtained.

Figure 7.4 compares the predicted speedup for Polyroots using processor sharing with the speedup of the original program (FIFO scheduling). With processor sharing, the speedup improves

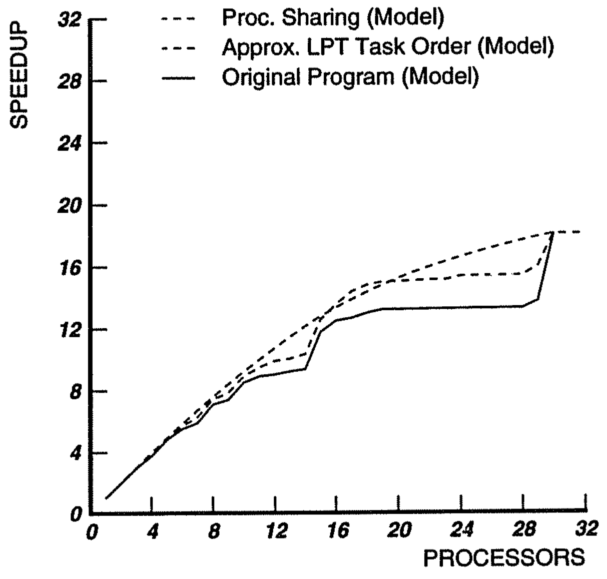


Figure 7.4. Processor sharing using P_{max} processes in program polyroots.

for all $P < 30$, but the predicted improvement is significantly higher than the Approximate LPT task order only for a few values of P (e.g., 14 and 29). In practice, of course, the overheads incurred in any system that provides processor sharing of tasks will not be negligible. Nevertheless, these results further demonstrate the modeling power of the task-graph-based analytical model.

It is also interesting to compare the predicted speedup to the processor sharing $\langle A + P_{\max} | PS \rangle$ bounds of Eager et al, and we show these curves in Figure 7.4 as well. The lower bound is much closer to the actual speedup in this case. Also, the bounds provide more accurate qualitative insight into the smooth speedup behavior of processor sharing than into the non-smooth performance of FIFO task scheduling.

7.2. Evaluating a Possible Design Change to PSIM

The speedup curve for PSIM in Figure 6.2 showed that the application has a speedup of about 13.5 at $P = 16$, and fairly poor speedups at small odd values of P. To analyze these results further, Figure 7.5 shows the three principal components of lost CPU cycles in this program for different values of P, as predicted by the deterministic model. Each bar shows the average fraction of execution time lost to communication (the components due to latency and bus contention are shown separately) and to idle time, each averaged across all the processors. The high average parallelism,

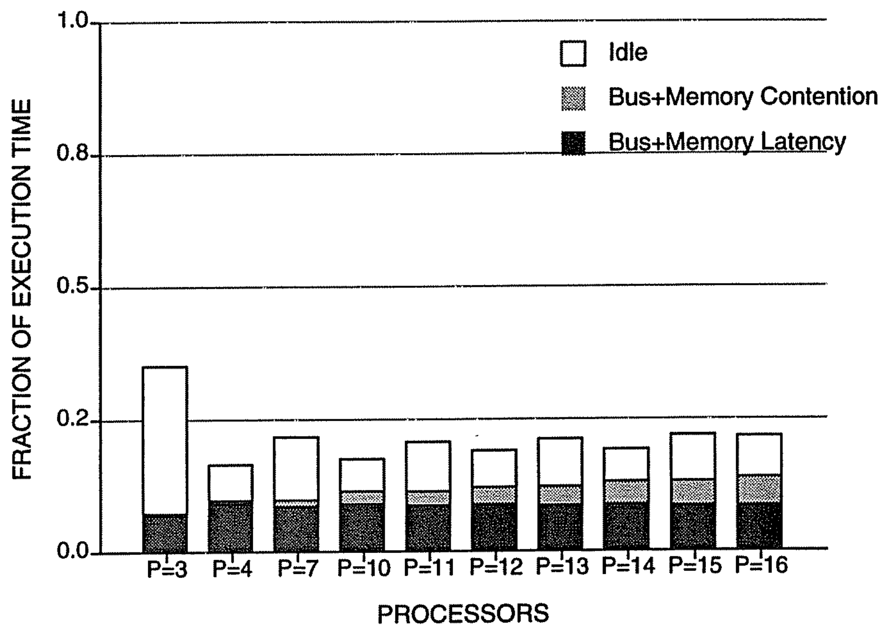
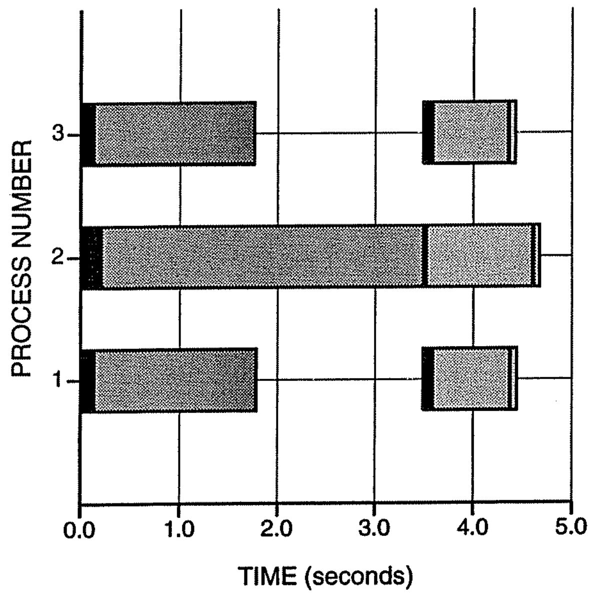
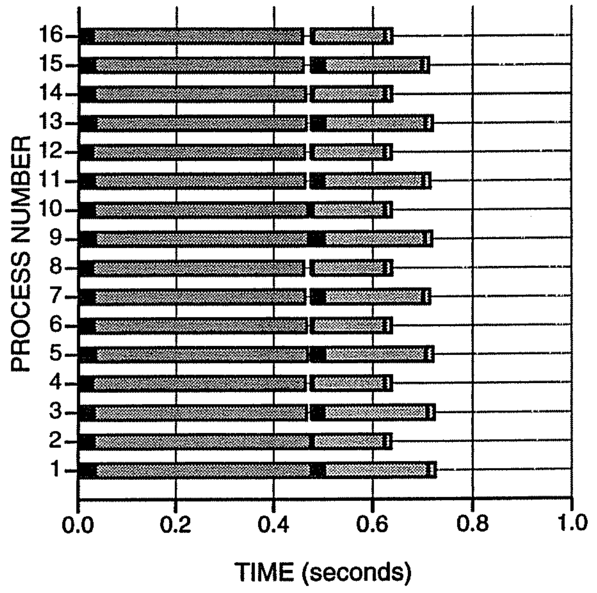


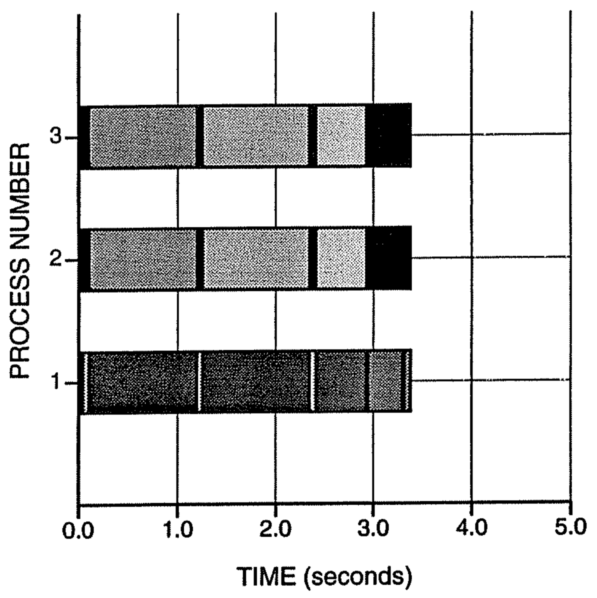
Figure 7.5. Sources of inefficiency in PSIM with processor-splitting task scheduling.
Input: 4096 node system; $N = 40963$, $A = 3407$

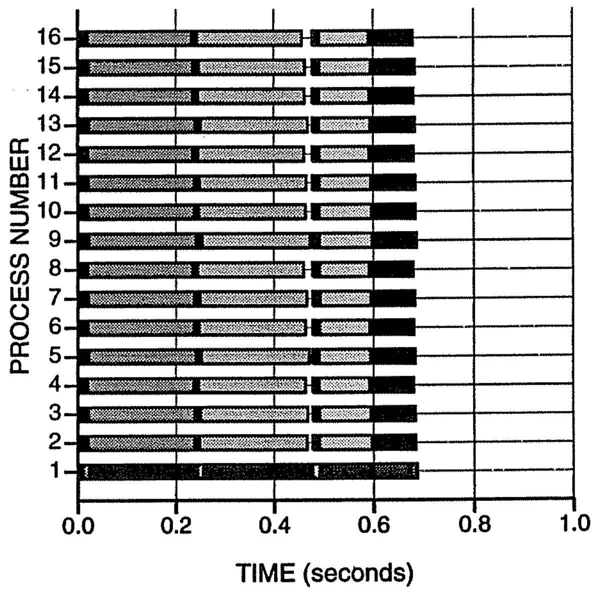
as well as the lower idle fraction at $P = 16$ than at $P = 3$, indicates that the idle cycles are caused by load-imbalance rather than insufficient parallelism. This is confirmed in Figure 7.6 (a), which gives predicted timelines of process execution in two cases, namely $P = 3$ and $P = 16$. In this graph, each interval of fixed shade represents the execution of a set of tasks belonging to a single loop (recall that each parallel phase of the program contains 6 parallel loops). The load-imbalance due to the processor-splitting scheduling method (which schedules different sets of loops on even and odd numbered processors in each phase) can be clearly seen for $P = 3$. More unexpectedly, there is also some load-imbalance in the second phase of the program at P=16: even though the same number of processors are assigned to every loop, the two sets of loops for even and odd numbered processors appear to contain unequal amounts of work.





(a) Processor-splitting task scheduling





(b) No processor-splitting

Figure 7.6. Process timelines showing executions of task groups in PSIM.

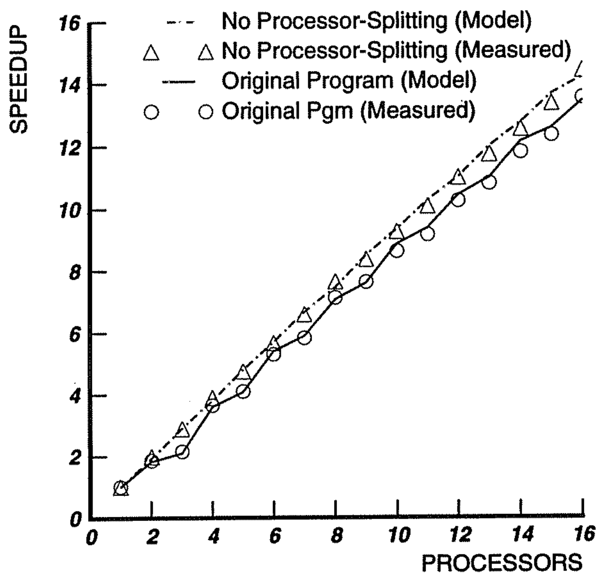


Figure 7.7. Improvement in speedup of PSIM without splitting processors between loops.

Since the tasks within each loop have approximately equal work, the load-imbalance should be smaller if, instead of splitting the processors between loops, the iterations of each loop were statically scheduled across all processors. The deterministic model can again be used to predict the execution times that would be obtained in this case. The predicted process timelines with no processor splitting for $P = 3$ and $P = 16$ are shown in Figure 7.6 (b). They show that this ordinary form of static loop scheduling achieves good load balance, at least at these values of P, with only slight imbalance in the first phase at $P = 16$. We also modified the program code to eliminate processor-splitting and directly measured the new execution times. In Figure 7.7, we compare the predicted and measured speedups for the original as well as the new scheduling method. The predicted improvement in speedup is about 5.5% for even values of P, and as high as 40% for P=3. Once again, the deterministic model was able to accurately predict the achievable performance improvement: the graph shows that predicted performance with the suggested design change is very close to the performance attained by the subsequently modified code. We also used the deterministic model to examine dynamic scheduling of the loop iterations (ignoring any scheduling overhead such locking the index variables), but the predicted further improvement was negligible. Examining the components of lost CPU cycles with the new scheduling (Figure 7.8), we see that the lost cycles are now almost entirely due to communication overhead, including significant bus contention.

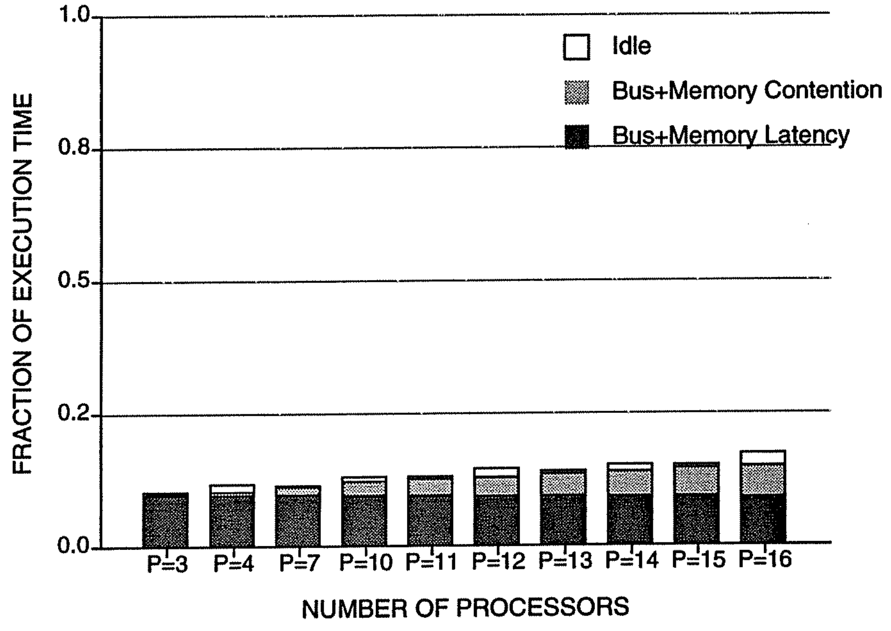


Figure 7.8. Sources of inefficiency in PSIM without processor-splitting.
Input: 4096 node system; $N = 40963$, $A = 3407$

7.3. Evaluating Communication Locality and Load Balancing in Locus Route

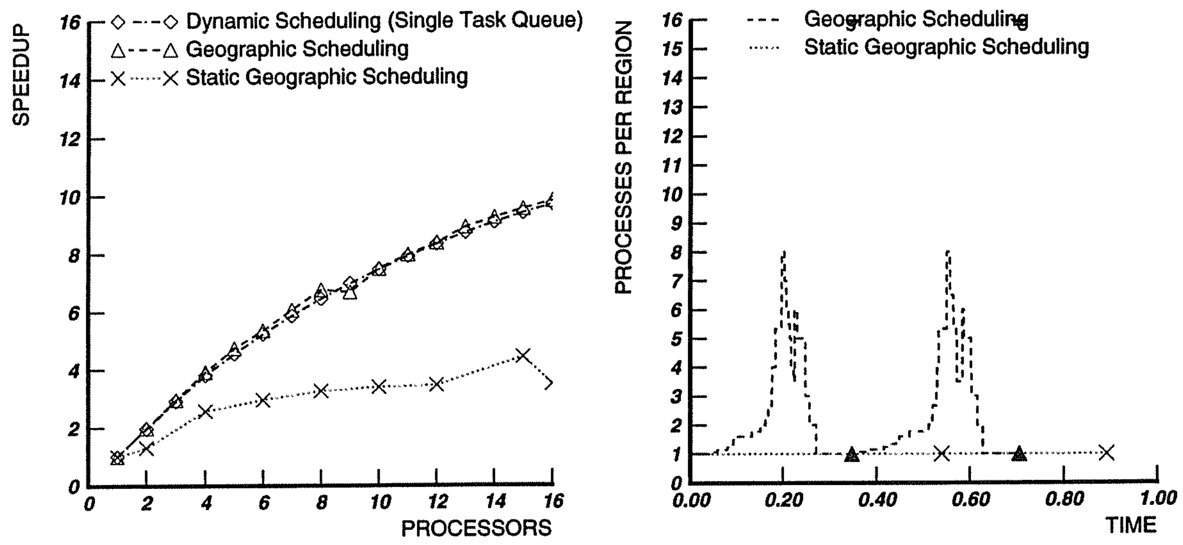
In some applications, a trade-off must be made between locality of communication and load-balancing of computational work: careful static task allocations can enhance locality but may cause load-imbalance, while dynamic scheduling can provide better load-balance but may reduce the locality of communication. Evaluating such design choices requires considering both effects. The deterministic model can predict communication costs in a program accurately, *given* the appropriate parameters such as cache miss rates. Unfortunately, when studying possible modifications to a scheduling algorithm and their impact on communication, *predicting* the effect of the modified scheduling on parameters such as cache miss rates can be difficult. Thus, in general, it may be necessary to measure these parameters, which in turn requires implementing the modified code. In fact, however, the deterministic model can be used *a priori* to obtain at least some task and process level information about communication locality, and thus to explore design choices that influence locality as well as load-balancing.

The VLSI wire routing program, Locus Route, requires a dynamic load balancing strategy because of the widely varying execution times of the individual tasks. However, two or more processes routing wires through overlapping regions of the chip must read and intermittently update common portions of the cost-array data structure, and this is the principal source of remote communication (cache misses) in the program [SWG92]. To reduce this communication requirement, Locus Route provides a semi-static task scheduling option called *geographic scheduling* in which separate task queues are maintained for tasks corresponding to wires in different regions of the chip. The underlying principle is to enhance processor locality of data access by allocating only one or perhaps a few processes per task queue. The chip is divided into a number of regions of equal area, with one task queue used for each region, and each wire placed in the task queue of the region containing its leftmost pin.[22] In each iteration, each processor is initially assigned to a single queue. To reduce load imbalance, a process takes tasks from another queue once its own queue becomes empty, choosing a queue with the fewest number of processors assigned to it.

Figure 7.9 (a) compares the predicted speedups of geographic scheduling and dynamic scheduling from a single task queue for the same input as before (bnrE.grin) on the Sequent Symmetry, using measured values of the communication parameters for each case, as in the experiments in Chapter 5. (For comparison, the figure also shows the speedup of the static form of geographic scheduling, i.e, where a process stays idle after completing its allocated tasks. To predict the speedup in this case, the measured communication parameters for geographic scheduling were used, since the static form is not actually implemented in the code. As expected, this scheduling method has poor performance due to extremely poor load balance.) The two dynamic policies have very similar speedups (measured as well as predicted), in part because the latency of communication on the Sequent system is relatively low and hence only a small fraction (about 4-5%) of execution time is lost to communication in each case. On systems with higher communication latencies relative to CPU speed, however, higher locality is likely to be more significant.

To explore to what extent this locality is compromised by the need for processes to switch task queues to preserve the load-balance, we used the model to compute how the average number of

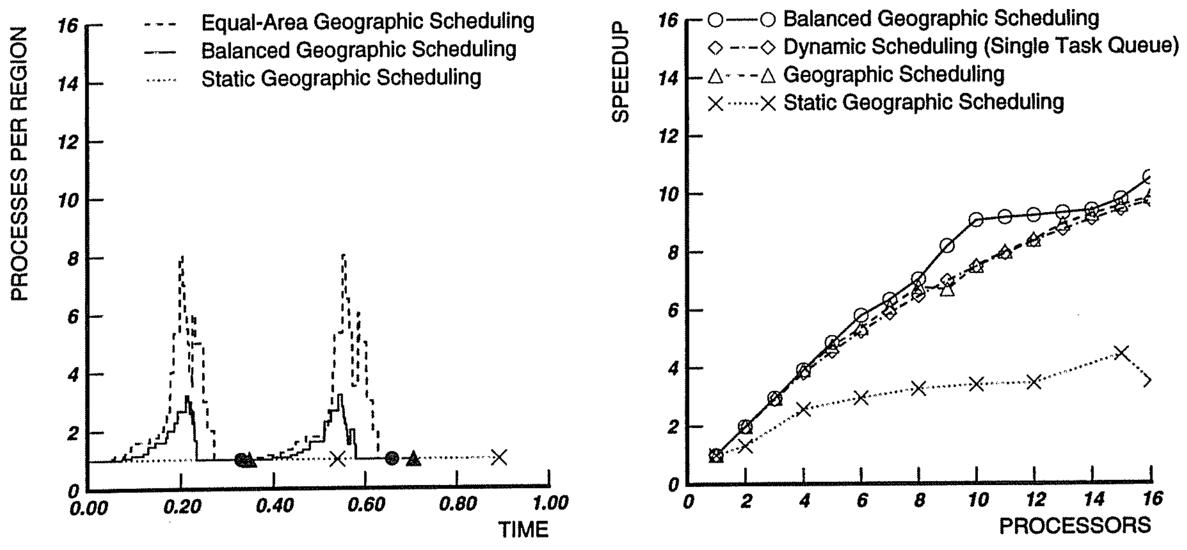
22. Since long wires can span multiple regions, processes working on different queues will still need to share data. Computing regions to ensure non-overlapping data access, even if possible, would require much more sophisticated computation.



(a) Speedups (b) Processes per region during an execution ($P = 16$)

Figure 7.9. Predicted impact of dynamic, semi-static and static scheduling in Locus Route.

active processes per active region changes over the interval of execution of the program. Figure 7.9 (b) plots this average as a function of time in an execution on 16 processors, using 16 regions for geographic scheduling. (The pair of points on each curve mark the end of the first and second iterations of the program.) In the static form of geographic scheduling, exactly one process works on each region throughout the execution. In the actual (i.e., non-static) geographic scheduling, however, the average changes as processes switch from empty to non-empty queues. (An increasing step in the figure indicates either that a process switched queues, joining a region that already had one or more processes working on it, or that the last process working on a region became idle, thus decreasing the number of active regions. A decreasing step indicates that a process became idle while at least one other process still had an unfinished task for the region.) The figure shows that some processes complete their allocated tasks and have to switch queues early in each iteration, indicating the some task queues contain relatively little work. For a substantial portion of each iteration, the average number of active processes per region is fairly large, indicating that a few queues contain a large fraction of the total work. Thus, the key conclusion from this figure is that an unbalanced initial division of work, i.e., causing some regions to contain much more work than others, could significantly compromise the processor locality of communication.



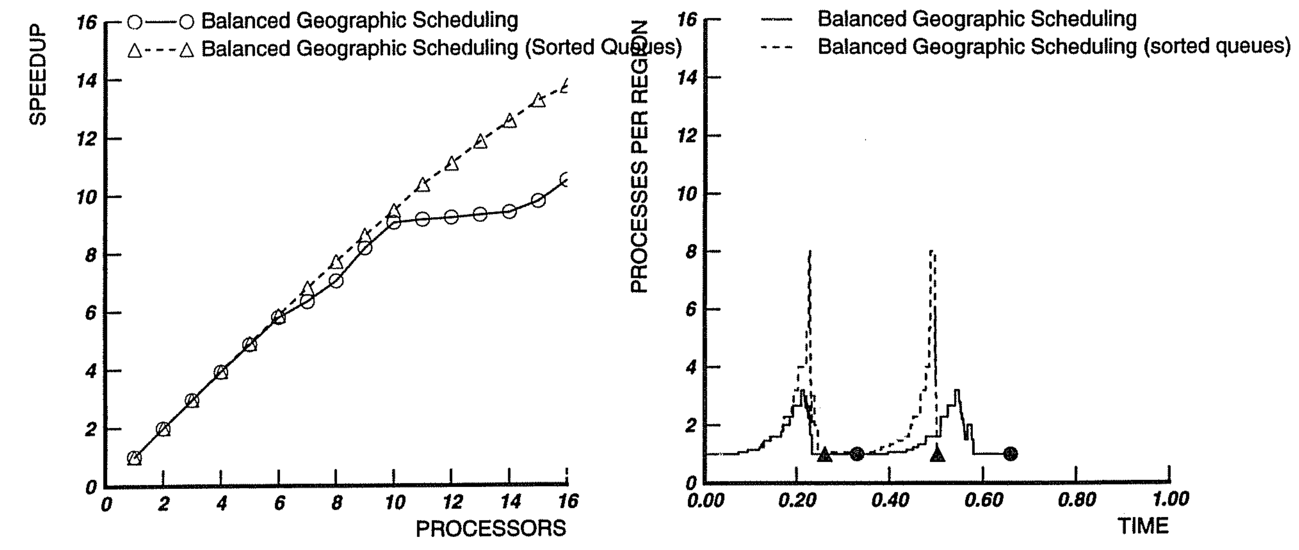
(a) Speedups (b) Processes per region during an execution ($P = 16$)

Figure 7.10. Predicted impact of balancing work across task queues with semi-static scheduling.

The above results suggest using a more balanced initial division of the chip into spatial regions. We used the deterministic model to study one hypothetical division of this type, in order to evaluate the possible effect on locality. To restrict the division to be based on information available to the program, we used the area of the bounding box of each wire (the smallest rectangle containing it) as a measure of the work required for the wire [SWG92]. We then divided the chip into rectangular regions containing approximately equal amounts of work, still using the leftmost pin of a wire to define its "location" on the chip. The division is approximate mainly because of the artificial restriction to rectangular regions, assumed in order to simplify the computation required to do the division. By specifying the new allocation of tasks to queues, the model can once again be used to predict the detailed evolution of the average number of processes per region with this new allocation. (For the low-level communication parameters, we used the values measured earlier for geographic scheduling. These could not be directly measured since the study was done entirely without modifying the program.) Figure 7.10 (a) compares this average with the corresponding curve for the original (i.e., "equal-area") geographic scheduling, shown previously. The figure shows that the improved division of tasks significantly reduces the length of time for which the average is greater than one and also significantly reduces the average number of processes per queue over substantial intervals of the program. Together, these curves indicate that the more equitable, yet simple, initial division of tasks should enhance the processor locality in the program.

We also found using the model that a balanced initial division of tasks among task queues incidentally tends to improve the overall load balance in the program, in addition to any improvement due to better communication locality. For example, *ignoring the change in communication locality* (since new values of the communication parameters are not available), Figure 7.10 (b) compares the speedups for the balanced geographic scheduling with those for the other scheduling alternatives shown earlier. (The communication parameters for equal-area geographic scheduling were used for the balanced geographic scheduling.) The figure shows that the balanced method has slightly higher speedups than the original geographic scheduling. This improvement is due to due to a slightly better order of execution of the wires. For example, on 16 processors, Figure 7.10 (a) shows that all but one of the tasks in each iteration complete well before the end of the iteration. In general, balanced geographic scheduling has a tendency to yield slightly better load-balance than scheduling methods that do not consider task execution times since large tasks will tend to begin execution earlier because of having fewer tasks in their task queue, on average. However, it could also perform significantly worse. In general, as was seen earlier in Polyroots, tasks with widely varying granularities can make the overall execution time sensitive to the order of execution of tasks.

The above discussion indicates that a heuristic reordering of the order of execution of tasks, as in the case of Polyroots, might be used to reduce the execution time. Using the same estimate of computation work as before, this can easily be combined with the balanced task allocation: after allocating the task to the different queues, the tasks in each queue can be reordered in decreasing order of computation work. As before, we used the deterministic model to predict the execution times with this change. (Unlike in Polyroots we are not predicting the effect of perfect LPT ordering based on *known* task execution times, but only the approximate ordering which can be implemented in the program.) Figure 7.11 (a) compares the new speedups with those for the unsorted balanced method and the original scheduling methods. The figure shows that the heuristic reordering based on approximate estimates of task execution times is successful in improving speedup, and provides significant improvement for $P > 10$. Furthermore, unlike the contingent improvements in load-balancing for the (unsorted) balanced geographic scheduling, our intuition about the benefits of LPT ordering indicates that the performance of this heuristic should be consistent for other input circuits as well (assuming the approximate estimates of task execution times are not less reliable).



(a) Speedups (b) Processes per region during an execution ($P = 16$)

Figure 7.11. Predicted impact of reordering tasks with balanced semi-static scheduling.

These improvements in load balance due to reordering, however, come at the possible cost of some decrease in locality compared to the unordered balanced geographic scheduling. Specifically, the key effect of initially sorting the queues is to increase the likelihood that processes that complete their work early will find leftover tasks in other queues. Consequently, at the end of each iteration, we can expect the average number of processes per queue to be higher than in the unsorted balanced case. Again, we can use the deterministic model to obtain some information about this loss in locality by predicting the average number of active processes per active region as a function of execution time under the reordered balanced geographic scheduling. Figure 7.11 (b) compares this average with the curve previously given for balanced geographic scheduling. As expected, we see a narrow spike in the former curve towards the end of each iteration. The interval over which this number is much larger than 1 is still small, however, and thus the locality is reduced for only a small portion of the execution time. (The figure also confirms the improved load-balance as shown by the steep drop in the number of active processes close to the end of each iteration.) Overall, we conclude from these results that the loss of communication locality due to the reordering of tasks is likely to be worthwhile, given the significant improvement in load-balance predicted by the model.

In the above experiments, metrics from the deterministic model suggested a modification to the scheduling in a program to improve locality of communication, provided some task and process level information about the potential improvement in locality, suggested a further improvement in load balancing and provided information about the consequent trade-off in locality. Thus, although new communication parameters such as cache miss rates are difficult to obtain, the model could 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 Locus Route, semi-static scheduling using multiple tasks queues is a typical method for improving communication locality while preserving acceptable load balance. Similar experiments would be possible in other such programs as well. In each case, however, a complete comparison of execution times under the various design alternatives would require knowledge of their impact on cache miss rates and other parameters describing inter-process communication. In particular, comparing such design alternatives analytically would require analytical techniques to derive these parameters for a given program, scheduling method and system configuration. If such techniques can be developed, the deterministic model would provide the framework to include these techniques in a complete evaluation of the above kind of program design alternatives as well. These issues are briefly explored further in Chapter 9.

7.4. Summary

To summarize the results of this chapter, we used the deterministic 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 deterministic model to precisely represent the important details of the task scheduling and execution of a program and to compute synchronization costs sufficiently accurately in each case. Section 5.2 showed that the principal defects in previous analytical models fall in exactly these two areas: the exponential task models yield inaccurate predictions of synchronization cost, while the simple i.i.d. task models are unable to capture details of task scheduling. Furthermore, only the full Markov chain models are able to distinguish between different orderings of tasks in a task queue, whereas the other models (including Mak and Lundstrom and Kruskal and Weiss) do not distinguish between different orderings. Thus, the deterministic model is able to overcome key limitations of previous models. The model can also be used to obtain task and process level information about communication locality that can be useful for comparing design issues that affect locality as well as load-balancing. Overall, we believe these results indicate that the model can support useful and potentially important performance prediction for parallel programs.

Chapter 8

Further Implications of the Study of Random Delays

The three previous chapters used the results from the study of random delays to motivate, develop and demonstrate an improved model for parallel program performance prediction. In this chapter, we briefly discuss other implications of the results of the random delays study. In particular, we use examples to show the implications for stochastic performance models of parallel systems (Section 6.1), and we briefly describe some insights the results provide for programmers of parallel systems (Section 6.2).

8.1. Implications for General Parallel Processing Models

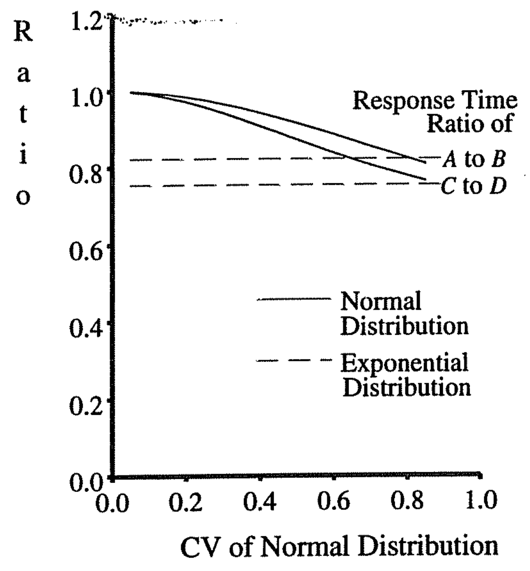
The data presented in Chapter 3 strongly indicate that the principal effect of communication delays in shared-memory parallel programs is to increase the mean completion time of a process in a phase, and not the variance, even under conditions of high communication costs and contention. Furthermore, the data indicate that the overall variance of execution time between synchronization points due to both communication delays and processing requirements is also extremely small in many programs. Both results were corroborated for the same programs by the evidence of Section 4.4: a deterministic model that ignores both sources of variance proved to be accurate for the programs studied.

Stochastic models are nevertheless important for the performance evaluation of high-level design issues in parallel systems including issues such as multiprogrammed multiprocessor scheduling policies, synchronization management policies within programs, and sometimes abstract models of parallel program behavior as well [ALL89, BaL90, ChN91, LeV90, LeN91, LCB92, NTT88, Nel90, NTT90, SeT91, ZaM90]. For example, the models in multiprocessor scheduling performance studies must represent complex workloads consisting of many different jobs (programs). Representing the characteristics (e.g., CPU demands, parallelism, etc.) of each individual workload member would be too tedious and sometimes impractical. In such cases, a stochastic representation of aggregate workload characteristics is usually necessary.

Many parallel system performance models must assume exponentially distributed task or process execution times to permit tractable analysis. The evaluation of stochastic models for parallel programs in Chapter 5 might seem to indicate that parallel system models that assume high variance are seriously limited as well. This extrapolation is not valid because the goals of the two types of models are typically quite different. In particular, for the models in Chapter 5, numerical accuracy of the predictions for a specific program is an important consideration. In many parallel system performance models, however, the nature of the problem requires a qualitative comparison of design alternatives based on quantitative performance estimates. More generally, the qualitative or quantitative conclusions of the models may not be sensitive to the precise representation of synchronization costs in individual programs. Our results in Chapter 3 emphasize that it is important to determine by informal reasoning or formal validation how the model results depend on the exponential assumption. Specifically, the results imply that if a conclusion of such a model would be significantly weaker, or even different, for programs that have much lower task time variance, it may not apply to many parallel programs. In the remainder of this chapter, we discuss two examples of previous results obtained through stochastic performance models of parallel systems. The conclusion in the first example does not appear to be strongly influenced by the distributional assumption used to derive it, whereas in the second example the strength of the conclusion is affected by the choice of distribution used to represent the parallel workload.

A result that is not strongly dependent on the distribution assumption is as follows. Nelson et al [NTT88] showed that in an environment containing mixed sequential (interactive) and large parallel (batch) jobs, an unpartitioned parallel system yields better performance than one in which processors are statically partitioned among the two classes. They assumed that the parallel jobs consisted of tasks with exponentially distributed execution times. But, in fact, Setia and Tripathi [SeT91] showed that the same conclusion holds with a completely different assumption about task times. Specifically, they assumed that *each job consisted of tasks of equal size*, while the total execution time of the jobs on any fixed number of processors was assumed to be exponentially distributed.

In contrast, one result that is significant for exponentially distributed tasks but weaker for tasks with lower variance is Nelson's result [Nel90] that higher variance of parallelism can yield



Class	Dist.	Mean $\bar{\beta}$
A	$\beta_1 = 0.8$	2.4
	$\beta_8 = 0.2$	
B	$\beta_2 = 0.6$	2.4
	$\beta_3 = 0.4$	
C	$\beta_1 = 0.5$	16.5
	$\beta_{32} = 0.5$	
D	$\beta_{15} = 0.5$	16.5
	$\beta_{16} = 0.5$	

Figure 8.1. Comparing predictions from Exponential and Normal Distributions.

lower response times, when queueing effects are small. He considers a parallel processing system with P processors and an arrival stream of parallel jobs, where an arriving job splits into n tasks with probability β_n, $1 \le n \le N_{\max}$. Each task has exponentially distributed processing requirement with unit mean. He compares two types of jobs, one with higher variance of parallelism than the other, and shows that the jobs with higher variance have significantly lower response times when queueing effects are negligible. This result arises because the completion time of n tasks on P processors is a concave increasing function of n. (For example, for exponentially distributed task times with mean $1/\mu$, and for $n \le P$, it is given by $(1/\mu)\sum_{i=1}^{i=n} 1/i$.) Although this result holds for any concave task time distribution, it is weaker when the curve grows more slowly as a function of n, such as for distributions with lower variability. To show the effect of the choice of distribution, consider the two systems (A and B) that were compared in [Nel90], each with 8 processors but different distributions of parallelism (β_i) as given in Figure 8.1. Both systems have mean parallelism of 2.4, but A has a higher variance of parallelism than B. If each task has exponentially distributed execution time, the ratio of the mean response times of A and B is about 0.82 (i.e., A has 18% lower response time). Now consider the same two systems, but assume task execution times are normally distributed with unit mean and variance σ^2. In Figure 8.1, we plot the ratio of average response times of A and B for a range of values of CV = σ. The ratio is close to 1 for low variance and approaches approximately 0.82 as σ gets close to 1. We repeat this comparison for another pair of systems, C and D, with 32 processors each, and C having a higher variance of parallelism. The performance ratio of C to D is about 0.75 with exponentially distributed task times, but it is again close to 1 for low σ. Thus, jobs with higher variance of parallelism do show lower response times, but the effect is significant only when the variance of task execution time is high.

The above example is intended only to emphasize that it is important to evaluate the effects of distribution assumptions on the results of a model, as we stated earlier. If a result is dependent on a task time distribution with high variance, our findings show that it may not apply to many parallel programs.

8.2. Implications for Programmers

Load balancing is an important aspect of the design of efficient parallel programs. In particular, programmers have to choose between static and dynamic load balancing for probably every parallel program. Static load balancing is an attractive choice because it is simpler to implement and debug. In most systems today, static scheduling performs well if the computation load is evenly balanced. In future systems with much higher communication costs relative to computational speed, static scheduling would not perform well if contention introduced significant variability in task completion times, even when the computational work is divided evenly among the processes. Our results indicate that this problem is not likely to arise in practice, and the variance introduced by random communication delays can usually be ignored when choosing between static and dynamic scheduling.

It is important to emphasize here that the above argument refers solely to the *variance* introduced by communication and contention delays. In particular, the mean communication costs of the individual processes sometimes cannot be ignored in choosing a task scheduling policy. For example, locality of communication may be compromised by dynamic load-balancing. Even with static scheduling, a policy that ensures a degree of processor affinity for tasks accessing common data may provide better performance than one that does not ensure such affinity. Nevertheless, these observations do not contradict our conclusion above: for any particular scheduling policy, the load-balance will not be significantly affected by the variance introduced by communication and contention delays. In other words, this variance can be ignored when choosing between different scheduling policies, as stated above.

Chapter 9

Summary and Directions for Future Research

Parallel processing systems hold out the promise of enormous computing power and, perhaps more important, cost-effective computing power across a range of performance levels from high-performance workstations to large-scale multiprocessors. To meet this promise, however, qualitative improvement in the programmability of parallel systems will be necessary. I believe two complementary advances are essential before truly general-purpose parallel programming becomes prevalent. The first is an abstract parallel processing model that can be supported efficiently in hardware and can also serve as a practical model for the design and analysis of parallel algorithms and software. A deep understanding of the important qualitative aspects of parallel processing will be essential for the development of such a model. The second is a comprehensive set of techniques for quantitative performance prediction that will enable programmers and algorithm designers to evaluate design choices and trade-offs. This thesis has shown that analytical performance evaluation techniques can contribute significantly to the understanding of parallel program behavior and provide useful, efficient techniques for quantitative performance prediction.

9.1. Contributions of the Thesis

The first part of this thesis developed an analytical model to study one potentially important aspect of program behavior, namely the non-deterministic nature of delays due to inter-process communication and shared resource contention. This model yields the key insight that even if the individual delays during a process's execution are highly variable, the total delay over an interval of execution will have small variability if the interval includes a sufficiently large number of these individual delays. Detailed measurements show that in shared-memory programs on current systems, the number of individual communication delays in intervals between synchronization points is sufficiently large that such delays introduce negligible variance into the execution time of a process between successive synchronization points, even under conditions of high communication cost and contention. Furthermore, extrapolation based on the analytical model indicates that this conclusion will continue to hold for systems in the foreseeable future, at least for shared-memory programs with granularities similar to those on current systems. Finally, for many but not all such programs, non-deterministic processing requirements also introduce only small variance into the execution time between synchronization points.

These results hold potentially useful insights for programmers as well as designers of shared-memory parallel systems. In addition to helping programmers reason about parallel program behavior, these results can provide guidance in specific design decisions. They indicate, for example, that when making design choices for load balancing, a programmer need only consider processing requirements and mean communication costs, and can ignore the variability in communication costs. For analytical models that are used for parallel program performance prediction, the results imply that it may be reasonable to ignore the variance of process execution times when computing synchronization costs. In particular, a deterministic performance prediction model may be more accurate than one that assumes high variability in task times, such as in the exponential distribution. For stochastic performance models of parallel systems, especially models that assume exponentially distributed task or process execution times, the results indicate that it is important to determine whether model conclusions are sensitive to the assumed task or process execution time distribution: such conclusions may not be accurate for many parallel program workloads. Finally, by contributing to our understanding of a basic aspect of parallel program performance, the above results may contribute to the development of a useful and general model of parallel processing, referred to above.

The second part of this thesis developed, validated and demonstrated the use of a deterministic analytical model for parallel program performance prediction. The choice of using a deterministic model was motivated both by the results of the above study of random delays and by a qualitative assessment of the limitations of previous stochastic models. Specifically, these previous models are either restricted to programs with simple task graphs or require complex and non-intuitive solution techniques as well as the assumption of exponential task times to model programs with more complex task graphs. The deterministic model developed in this thesis instead allows a straightforward and intuitive solution technique for arbitrary task graphs and a wide range of static, semi-static and dynamic task scheduling disciplines. Experimentally, it has proved efficient and accurate for the set of programs tested. In contrast, experimental data provided in this thesis shows that stochastic models that only apply to simple task graphs and scheduling disciplines have inconsistent accuracy (when they apply), whereas models allowing more sophisticated graphs and scheduling proved to be extremely inefficient and often inaccurate. The inaccuracy in the latter models is principally due to the assumption of exponential task times. Also in comparison, parametric speedup bounds and estimates require parameters that are not significantly easier to obtain in practice than the solution of the deterministic model. Furthermore, as shown in Chapter 6, these parametric techniques do not provide significant qualitative information that is available in many cases from the deterministic model, and, perhaps most important, do not apply to non-work-conserving scheduling disciplines including common ones based on static scheduling.

The thesis also provided evidence that the deterministic model can be useful for predicting, understanding and improving certain aspects of program performance. By using an abstract representation of the inherent parallelism structure and task scheduling, the model provides the ability to quickly predict program performance on varying system sizes, varying input sizes, and with a variety of task scheduling strategies. Performance metrics related to the task graph and scheduling are sufficiently detailed to allow a programmer to quantify and understand nuances of program performance. A key aspect of the model is that it represents the program at the task level, where (informally) a task is an independent unit of sequential work. We believe that this identification of tasks, defined in Section 2, is appropriate for reasoning about high-level design decisions during program development, particularly decisions about task granularity, partitioning and scheduling that affect load-balancing in a program.

9.2. Directions for Future Research

Overall we believe the deterministic model provides a promising basis for accurately and efficiently evaluating and predicting parallel program performance. The validations and performance studies in previous chapters have relied on one common methodology for obtaining model inputs to apply the model, specifically deriving the task graph by hand and measuring task execution times and resource usage parameters explicitly. They have also focused on one important class of parallel programs, namely computationally intensive scientific and engineering programs. These experiments and results suggest several directions for future research, including research to widen the scope of performance studies that can be carried out using the model, as well as to investigate using the model in new domains such as I/O intensive parallel applications. The latter case also provides scope for further uses of the renewal model to explore program behavior in these new domains. These avenues for future research are briefly discussed below.

The nature and scope of performance studies based on the deterministic model could potentially be expanded in two different directions: towards more abstract algorithm analysis similar to studies that use abstract computational models, and towards detailed and comprehensive program performance analysis in conjunction with measurement and simulation, incorporated into automated performance tools.

First, it would be interesting to investigate applying the model at an abstract level for analyzing and comparing algorithms, represented by their task graphs. Standard sequential algorithm analysis techniques might be used to estimate task CPU times, whereas a parallel computation model must also be considered for estimating the inherent communication requirements with other tasks. Finally, each of these inputs must be appropriately translated into numerical values that can be used by the model. With these inputs, the model could be directly applied to obtain numerical estimates of execution time, and thus used to study algorithm performance as a function of input size and number of processors, and to compare alternative algorithms. Although the model would not yield analytic expressions for the execution time of a given algorithm, this might be offset by the ability of the model to analyze complex algorithms (e.g, complex task graphs) and complex task scheduling policies, including dynamic policies. Finally, experience with using the deterministic model for algorithm analysis as well as parallel program performance prediction could contribute to the development of more sophisticated and comprehensive algorithm analysis models in the long run.

Second, it would be worthwhile to investigate incorporating the model into automated measurement or simulation-based performance analysis tools. If the process of deriving model inputs can be partially or fully automated, it would considerable simplify the use of the model for the programmer. Furthermore, measurement or simulation-based tools could use the deterministic model to provide task-level analysis of program performance. This would combine the complementary strengths of analytical studies and measurement or simulation, and could yield a number of benefits for each. The task-level analyses using the model (besides being simpler for the programmer to carry out) could be supplemented with more detailed performance experiments made possible by measurement or simulation, for example, analysis of communication events at the level of individual data-structures or code statements. Conversely, measurement or simulation tools should benefit from the efficiency and insight provided by the analytical model. Perhaps more important, such tools, which have generally been restricted to a given program on a specific system, could exploit the predictive power of the analytical model to explore the effect of system and program design changes. Overall, exploring the benefits afforded by the combination of these approaches would be a significant and interesting research problem. The initial and perhaps the more challenging research issues, however, will arise in developing the infrastructure required to automate the process of applying the model.

For the deterministic model to be integrated into automated performance tools, compiler or run-time support (and perhaps also programmer annotation) will be required for creating the task-graph of a given program. Some requisite infrastructure for program analysis is already available in parallelizing compilers such as Jade [RSL93] and others, and in run-time systems such as Chores [EaZ93]. In parallelizing compilers, the compiler automatically detects and enforces (a superset of) the data dependencies in a program, and implements the partitioning and scheduling of work. The task graph defined by these data dependencies can be directly constructed, together with the scheduling function, and instrumentation can be inserted to measure the other model parameters. In Chores, a program is specified in terms of atoms of work, and a collection of atoms that apply the same function to different data sets is called a *chore*. The atoms are always executed sequentially and thus correspond directly to the tasks of the program. The programmer can specify certain kinds of static and regular precedences between atoms (indexed chores) or implement more complex precedence structures by adding atoms to a chore during execution (dynamic chores). In the former case it should be possible to construct the task graph from the specified precedences between atoms.[23] Developing the necessary infrastructure for systems where it is not already available, as well as building on the infrastructure to implement the deterministic model would be significant research problems. Finally note that, besides simplifying the process of using the model for the programmer, a programming environment supporting automatic construction of model inputs might itself make use of the model. In parallelizing compilers, for example, the model might be used to guide various task granularity and scheduling decisions, especially decisions that can be made at compile-time.

23. Specifically, this should be possible at least in programs where no additional precedences are introduced using low-level synchronization constructs (with programmer annotation to indicate the absence of such constructs).

A related avenue for future research, one that could enhance the overall predictive power of the deterministic model, is to develop analytical techniques to predict cache miss rates and other communication parameters, or at least the changes in the parameters, for a given program on new system configurations and under new scheduling methods. Such techniques would enable the model to explore the impact of hypothetical system and algorithm design changes on communication costs in greater detail than is possible now. A preliminary step in this direction has recently been taken. Tsai and Agarwal [TsA93] describe a method for computing multiprocessor cache miss rates in shared-memory algorithms as a function of problem size, cache line size, and system size, for algorithms with simple, regular and fixed data reference patterns. However, their method requires a detailed analysis by hand of the communication pattern in the algorithm to derive the final miss rates. A new challenge is to develop an appropriate high-level representation of communication behavior from which the relevant low-level communication parameters can be derived. (This is analogous to the abstract, task-level representation of parallelism and scheduling, which makes it possible for the model to predict in detail the impact of hypothetical system and algorithm design changes on synchronization costs and load-balancing.) The appropriateness of the representation should be judged not only by the effort required of the algorithm designer for deriving it but also the insight into program behavior that can be directly gained in the process. At least for regular applications such as the ones studied by Tsai and Agarwal, we believe such a high-level model could be possible.

The validation and uses of the deterministic model in this thesis, as well as the studies of the effect of random delays, have focused on current computation-intensive scientific and engineering applications. Similar studies are possible for other classes of applications as well as for applications on future systems. First, the renewal model or extensions thereof could be used to examine the validity of the assumption of deterministic task and process execution times for new classes of programs. On future systems, even for the class of applications studied here, the conclusions of the study of random delays would have to be validated using measurements of the renewal model parameters for those systems. The results of such studies have impact both on the design of parallel programs and on the techniques used to evaluate and predict parallel program and system performance, as shown by the implications of the renewal model study in this thesis. Finally, either for new application classes or on new systems, if the deterministic task assumption appears reasonable, further experiments could explore whether the deterministic model continues to prove useful for exploring the key performance issues.

An important class of applications not considered in this thesis are I/O intensive applications. The validity of the deterministic assumption in the presence of significant I/O activity would have to be examined. Based on insight from the renewal model, we speculate that processes that perform large numbers of I/O operations should see low relative variability, even if the mean time for I/O is fairly high. In any case, directly applying the deterministic model for such programs should be an interesting test of the modeling approach. A key feature that could be found in such programs but was not present in the applications studied in this thesis is the use of multiple processes per processor to overlap computation with I/O activity. The accuracy of the model extensions for handling this case, suggested in Section 4.2.3, would have to be evaluated. If successful, the model could yield a potentially important analytical evaluation technique for such programs. For example, the model might be useful for application areas such as parallel query processing in database systems where, again, significant partitioning and scheduling issues arise [ShN93]. A common workload characteristic complicating the problem of partitioning is data skew. In such cases, skewed task execution times would have to be represented as a set of (unequal) deterministic quantities, with the partitioning of work represented by the scheduling function. These aspects are analogous to the task execution times and scheduling in the applications studied in this work, and it appears reasonable to explore this approach for the new class as well.

Though much is taken, much abides. Today, large-scale parallel processing systems are on the verge of the transition from research vehicles to mainstream use, but exploiting the computing power of these systems remains a complex and laborious task. Nevertheless, it appears realistic to look forward to the day when our understanding of parallel computation and the techniques and tools available for programming parallel systems can keep pace with the size and power of these systems, and when parallel programming is as commonplace as sequential programming is today.

Appendix A. Proof that F_T is Asymptotically Normal

Claim. If σ_P^2 and σ_C^2 are finite, then $(T(D)-\mu(D))/\sigma(D) \Rightarrow$ Normal(0,1) as $D \rightarrow \infty$, where $\mu(D)$ and $\sigma(D)$ are as given in (7).

Proof. The proof is essentially a direct application of the Central Limit Theorem for cumulative reward processes. For a renewal process generated by the i.i.d. sequence $\{X_i\}$ (with $E[X_1] = \mu$), let $\{W_i\}$ be a sequence of i.i.d. random variables (rewards), where W_i is independent of $\{X_j\colon j \neq i\}$. Define the cumulative reward $C(t)$ to be $C(t) \equiv \sum_{i=1}^{i=R(t)} W_i$, where $R(t)$ is the number of renewals up to time t (just as was defined in (4), Section 3.2.1.1). Then, if $E[X_j^2]$ and $E[W_j^2]$ are finite,

$$\frac{C(t) - \frac{1}{\mu} t\, E[W_1]}{\left[(t/\mu)\, \mathrm{Var}(W_1 - E[W_1] X_1/\mu)\right]^{1/2}} \Rightarrow \text{Normal}(0,1) \text{ as } t \rightarrow \infty.$$

(A slightly more general case is proved in [Wol89, p. 124].) In our model, let $X_i = P_i$, $W_i = C_i$. Then $\mu = \mu_P$, $E[W_1] = \mu_C$, and $C(D) = \sum_{i=1}^{R(D)} C_i$. Then, under the hypotheses of the claim, the above theorem directly applies, showing that $C(D)$ converges to a Normal with mean $D\mu_C/\mu_P$, and variance given by

$$\mathrm{Var}(C(D)) = \frac{D}{\mu_P} \mathrm{Var}(C_1 - \frac{1}{\mu_P} E[C_1] P_1)$$

$$= D\frac{\sigma_C^2}{\mu_P} + \frac{D{\mu_C}^2}{{\mu_P}^3}\sigma_P^2$$

But, $T = D + C(D)$, since $C(D)$ is just the total of the communication delays in the first $R(D)$ cycles. Thus T also converges to a Normal with mean $\mu_T = D + D\mu_C/\mu_P$ and variance σ_T^2 the same as the variance of $C(D)$ calculated above. **QED.**

Appendix B. System-Level Model for the Sequent Symmetry

In this appendix, we briefly describe the system-level model component we developed to calculate system overhead costs for applications on the Sequent Symmetry. For the applications with significant system overhead (MP3D, PSIM and Locus Route), remote communication was the most important source of such costs, and we used a queueing network model of the Sequent bus sub-system to calculate them. We ignored other sources of overhead, namely lock contention and forking overhead, for our study. If these costs are also significant, Tsuei and Vernon have shown that each can be separately and accurately included [TsV90].

The Sequent Symmetry bus supports an invalidation-based snooping cache protocol. In a previous analytical modeling study of the Sequent bus [TsV92], Tsuei and Vernon showed that two aspects of the bus protocol have a significant impact on performance: (1) at most three read requests can be outstanding at any time from all processors, with at most one per processor, and (2) responses to read requests have higher (non-preemptive) priority for the bus than all other bus requests. In their model, Tsuei and Vernon used a Markov chain in addition to a queueing network to represent these and other details of the bus protocol. We develop a much simpler but less detailed model which represents both the above features of the protocol using a queueing network alone. We compared the predictions of our model 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. The model is as follows.

The possible types of remote communication requests on the bus are *read*, *read+write_back* and *invalidate*, 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 active processor) to characterize remote communication behavior on the Sequent:

λ_{bus}	Mean request rate to bus per active processor
$f_{invalidate}$	Fraction of requests that are of type *invalidate*
$f_{r,wb}$	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 as the resource usage inputs to the model. (Thus, referring to Table 3.2, $N_{param} = 4$.) When solving the system-level model, the state *cur_state* specifies the number of active processors, p_{active}, as well the identities of the actively executing tasks. λ_{bus} is set to the average of $\lambda_{bus}(t)$ for all active tasks t, and the other three parameters are calculated similarly. (If more than 1 process is allocated to a processor, each active task must be weighted according to the fraction of processing power it receives.)

Our system-level model is a closed single-class queueing network model with P_{active} customers, and with the bus and the two memory modules represented as queueing centers, and the caches and processors represented as infinite-server (i.e. delay) centers. The queueing network is solved using the recursive MVA algorithm, with recursion on the customer population n from $n=1$ up to P_{active}. (Note that since P_{active} is the number of active *processors*, rather than processes or tasks, it is effectively just a small constant in this system. For larger systems, e.g., with a few hundred processors, a non-recursive MVA solution might be desirable.) The two important features of the bus protocol mentioned above are incorporated into this solution using heuristic approximations. The non-preemptive priority of the read responses is modeled using a standard MVA priority approximation (service-time-inflation) [BKL84]. The limit of three outstanding read requests is modeled using an additional delay center in the queueing network. For $n \geq 4$, read requests visit the delay center with probability p_{block} *before* using the bus, and the mean delay time per visit to the delay center is R_{block}. For each value $n \geq 4$, p_{block} and R_{block} are first estimated by solving a separate M/M/3//n queue. The mean service time in this queue is 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 (calculated from the queueing network solution at population value $n-1$).

The solution of this overall queueing network in state *cur_state* gives the average response time for remote communication, R. Then, we calculate the delay for each executing task t as $T_{delay}(t, T_{remain}(t), cur\_state) = T_{remain}(t) \times R \,/\, \lambda_{bus}(t)$. Thus, a single solution of the system-level model yields $T_{delay}(t, \cdots)$ for all executing tasks.

References

[AdV93] V. S. ADVE and M. K. VERNON, Performance Analysis of Mesh Interconnection Networks with Deterministic Routing, *IEEE Transactions on Parallel and Distributed Systems (to appear)*, July 1993.

[ASH88] A. AGARWAL, R. SIMONI, M. HOROWITZ and J. HENNESSY, An Evaluation of Directory Schemes for Cache Coherence, *Proc. 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, June 1988, 280-289.

[ALK90] A. AGARWAL, B. LIM, D. KRANZ and J. KUBIATOWICZ, APRIL: A Processor Architecture for Multiprocessing, *17th Annual International Symposium on Computer Architecture*, May 1990, 104-114.

[Amd67] G. M. AMDAHL, Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities, *AFIPS Conference Proceedings 30*(1967), 483-485.

[AIA90] H. H. AMMAR, S. M. R. ISLAM, M. AMMAR and S. DENG, Performance Modeling of Parallel Algorithms, *Proc. 1990 International Conference on Parallel Processing*, 1990, III 68-71.

[BaL92] T. BALL and J. R. LARUS, Optimally Profiling and Tracing Programs, *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, January 1992, 59-70.

[Bro88a] E. D. BROOKS III, PCP: A Parallel Extension of C that is 99% Fat Free, Computational Physics Division Technical Report, Lawrence Livermore National Laboratory, September 1988.

[Bro88b] E. D. BROOKS III, The indirect *k*-ary *n*-cube network for a vector processing environment, *Parallel Computing 6*(1988), 339-348.

[BKL84] R. M. BRYANT, A. E. KRZESINSKI, M. S. LAKSHMI and K. M. CHANDY, The MVA Priority Approximation, *ACM Trans. on Computer Systems 2*, 4 (November 1984), 335-359.

[CMM88] R. C. COVINGTON, S. MADALA, V. MEHTA, J. R. JUMP and J. B. SINCLAIR, The Rice Parallel Processing Testbed, *Proc. 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1988, 4-11.

[CKP93] D. CULLER, R. KARP, D. PATTERSON, A. SAHAY, K. E. SCHAUSER, E. SANTOS, R. SUBRAMONIAN and T. EICKEN, LogP: Towards a Realistic Model of Parallel Computation, *Proc. Fifth ACM SIGPLAN Notices Symposium on Principles and Practices of Parallel Programming*, May 1993.

[Cve87] Z. CVETANOVIC, The Effects of Problem Partitioning, Allocation and Granularity on the Performance of Multiple-Processor Systems, *IEEE Trans. on Computers C-36*, 4 (April 1987), 421-432.

[DuB82] M. DUBOIS and F. A. BRIGGS, Performance of Synchronized Iterative Processes in Multiprocessor Systems, *IEEE Trans. on Software Engineering SE-8*, 4 (July 1982), 419-431.

[EZL89] D. L. EAGER, J. ZAHORJAN and E. D. LAZOWSKA, Speedup versus Efficiency in Parallel Systems, *IEEE Trans. on Computers C-38*, 3 (March 1989), 408-423.

[EaZ93] D. L. EAGER and J. ZAHORJAN, Chores: Enhanced Run-Time Support for Shared-Memory Parallel Computing, *ACM Trans. on Computer Systems 11*, 1 (February 1993), 1-32.

[FuK92] Y. O. FUENTES and S. KIM, Foundations of Parallel Computational Microhydrodynamics : Communication Scheduling Strategies, *A.I.Ch.E. J. 38*(1992), 1059-1078.

[Gre89] A. GREENBAUM, Synchronization Costs on Multiprocessors, *Parallel Computing 10*(1989), 3-14.

[HaM92] F. HARTLEB and V. MERTSIOTAKIS, Bounds for the Mean Runtime of Parallel Programs, *Proceedings of the Sixth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, September 1992, 197-210.

[HeT83] P. HEIDELBERGER and K. S. TRIVEDI, Analytic Queueing Models for Programs with Internal Concurrency, *IEEE Trans. on Computers C-32*, 1 (Jan. 1983), 73-82.

[HoS84] E. HOROWITZ and S. SAHNI, *Fundamentals of Computer Algorithms*, Computer Science Press International, Inc., Rockville, Maryland, 1984.

[KME89] A. KAPELNIKOV, R. R. MUNTZ and M. D. ERCEGOVAC, A Modeling Methodology for the Analysis of Concurrent Systems and Computations, *Journal of Parallel and Distributed Computing 6*(1989), 568-597.

[KSR91] KENDALL SQUARE RESEARCH, *KSR1 Principles of Operation*, Kendall Square Research, Waltham, MA, October 1991.

[KiS90] J. KIM and A. C. SHAW, An Experiment on Predicting and Measuring the Deterministic Execution Times of Parallel Programs on a Multiprocessor, Technical Report 90-09-01, Department of Computer Science and Engineering, University of Washington, September 1990.

[KrW85] C. P. KRUSKAL and A. WEISS, Allocating Independent Subtasks on Parallel Processors, *IEEE Trans. on Software Engineering SE-11*, 10 (October 1985), 1001-1016.

[Lar93] J. R. LARUS, Loop-Level Parallelism in Numeric and Symbolic Programs, *IEEE Trans. on Parallel and Distributed Systems 4*, 7 (July 1993), 812-826.

[LeN91] S. T. LEUTENEGGER and R. D. NELSON, Analysis of Spatial and Temporal Scheduling Policies for Semi-Static and Dynamic Multiprocessor Environments, *IBM Research Report*, August 1991.

[LCB92] G. LEWANDOWSKI, A. CONDON and E. BACH, Realistic Analysis of Parallel Dynamic Programming Algorithms, Computer Sciences Technical Report #1116, University of Wisconsin-Madison, Oct. 1992.

[MaS91] S. MADALA and J. B. SINCLAIR, Performance of Synchronous Parallel Algorithms with Regular Structures, *IEEE Trans. on Parallel and Distributed Systems* 2, 1 (January 1991), 105-116.

[MaL90] V. W. MAK and S. F. LUNDSTROM, Predicting Performance of Parallel Computations, *IEEE Trans. on Parallel and Distributed Systems 1*, 3 (July 1990), 257-270.

[MCH90] B. P. MILLER, M. CLARK, J. K. HOLLINGSWORTH, S. KIERSTEAD, S. LIM and T. TORZEWSKI, IPS-2: The Second Generation of a Parallel Program Measurement

System, *IEEE Trans. on Parallel and Distributed Systems 1*, 2 (April 1990), .

[Moh84] J. MOHAN, *Performance of Parallel Programs: Model and Analyses*, Ph.D. Thesis, Carnegie Mellon University, July 1984.

[NTT88] R. NELSON, D. TOWSLEY and A. N. TANTAWI, Performance Analysis of Parallel Processing Systems, *IEEE Trans. on Software Engineering 14*, 4 (April 1988), 532-540.

[Nel90] R. NELSON, A Performance Evaluation of a General Parallel Processing Model, *1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems 18*, 1 (1990), 14-26.

[PoK87] C. D. POLYCHRONOPOLOUS and D. J. KUCK, Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers, *IEEE Trans. on Computers C-36*, 12 (Dec. 1987), .

[RAM92] D. A. REED, R. A. AYDT, T. M. MADHYASTHA, R. J. NOE, K. A. SHIELDS and B. W. SCHWARTZ, An Overview of the Pablo Performance Analysis Environment, Technical Report, University of Illinois, November 1992.

[RHL93] S. K. REINHARDT, M. D. HILL, J. R. LARUS, A. R. LEBECK, J. C. LEWIS and D. A. WOOD, The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers, *Proc. 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993, 48-60.

[RSL93] M. RINARD, D. J. SCALES and M. S. LAM, Jade: A High-Level Machine Independent Language for Parallel Programming, *Computer 26*, 6 (June 1993), 28-38.

[Sar89] V. SARKAR, Determining Average Program Execution Times and their Variance, *Proc. 1989 SIGPLAN Notices Conference on Programming Language Design and Implementation*, 1989, 298-312.

[Seq81] SEQUENT COMPUTER SYSTEMS, INC., *Symmetry Technical Summary*, Sequent Computer Systems, Inc., 1988.

[SeT91] S. SETIA and S. K. TRIPATHI, An Analysis of Several Processor Partitioning Policies for Parallel Computers, University of Maryland CS-Tech. Rep.-2684, May 1991.

[Sev89] K. C. SEVCIK, Characterizations of Parallelism in Applications and Their Use in Scheduling, *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems 17*, 1 (May 1989), 171-180.

[ShN93] A. SHATDAL and J. F. NAUGHTON, Using Shared Virtual Memory for Parallel Join Processing, Computer Sciences Technical Report #1139, Univ. of Wisconsin-Madison, March 1993.

[Sha90] A. C. SHAW, Deterministic Timing Schema for Parallel Programs, Technical Report 90-05-06, Department of Computer Science and Engineering, University of Washington, May 1990.

[SWG92] J. P. SINGH, W. WEBER and A. GUPTA, SPLASH: Stanford Parallel Applications for Shared-Memory, *Computer Architecture News 20*, 1 (March 1992), 5-44.

[TaV85] R. E. TARJAN and U. VISHKIN, An Efficient Parallel Biconnectivity Algorithm, *SIAM Journal of Computing 14*, 4 (1985), 862-874.

[ThB86] A. THOMASIAN and P. F. BAY, Analytic Queueing Network Models for Parallel Processing of Task Systems, *IEEE Trans. on Computers C-35*, 12 (December 1986), 1045-1054.

[TRS90] D. TOWSLEY, G. ROMMEL and J. A. STANKOVIC, Analysis of Fork-Join Program Response Times on Multiprocessors, *IEEE Trans. on Parallel and Distributed Systems 1*, 3 (July 1990), .

[Tri82] K. S. TRIVEDI, *Probability and Statistics with Reliability, Queueing and Computer Science Applications*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[TsA93] J. TSAI and A. AGARWAL, Analyzing Multiprocessor Cache Behavior Through Data Reference Modeling, *Proc. 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1993.

[TsV90] T. TSUEI and M. K. VERNON, Diagnosing Parallel Program Speedup Limitations Using Resource Contention Models, *Proc. 1990 International Conference on Parallel Processing*, 1990, II 185-189.

[TsV92] T. TSUEI and M. K. VERNON, A Multiprocessor Bus Design Model Validated by System Measurement, *IEEE Trans. on Parallel and Distributed Systems 3*, 6 (November 1992), 712-727.

[TuG89] A. TUCKER and A. GUPTA, Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors, *Proc. 12th ACM Symposium on Operating Systems Principles*, December 1989, 159-166.

[VLZ88] M. K. VERNON, E. D. LAZOWSKA and J. ZAHORJAN, An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols, *Proc. 15th International Symposium on Computer Architecture*, June 1988.

[VSS88] D. F. VRSALOVIC, D. P. SIEWIOREK, Z. Z. SEGALL and E. F. GEHRINGER, Performance Prediction and Calibration for a Class of Multiprocessors, *IEEE Trans. on Computers 37*, 11 (Nov. 1988), 1353-1365.

[WiE90] D. L. WILLICK and D. L. EAGER, An Analytic Model of Multistage Interconnection Networks, *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1990, 192-202.

[Wol89] R. W. WOLFF, *Stochastic Modeling and the Theory of Queues*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1989.

[YaV91] N. YAZICI-PEKERGIN and J. VINCENT, Stochastic Bounds on Execution Times of Parallel Programs, *IEEE Trans. on Software Engineering 17*, 10 (October 1991), 1005-1012.