An Integrated Approach to Programming and Performance Modeling of Multicomputers

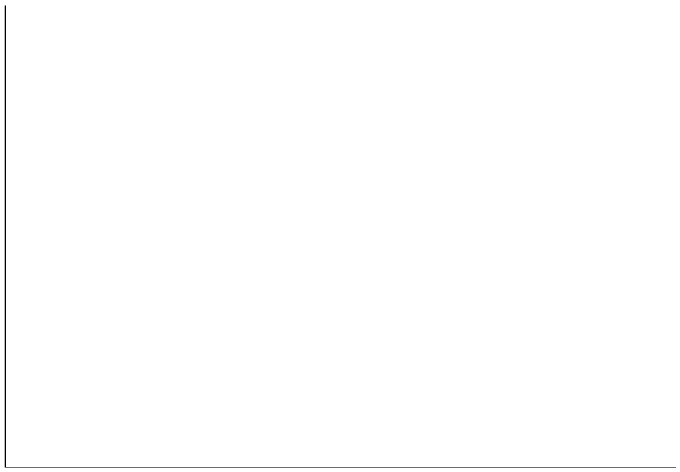
by

HALSUR V. SREEKANTASWAMY
B.Engg., University of Mysore, India, 1981
M.Tech., Indian Institute of Technology, Kanpur, India, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

IN THE FACULTY OF GRADUATE STUDIES
DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming to the required standard



THE UNIVERSITY OF BRITISH COLUMBIA

October, 1993

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science
The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1Z1

Date: Oct. 29, 1993

Abstract

The relative ease with which it is possible to build inexpensive, high-performance multicomputers using regular microprocessors has made them very popular in the last decade. The major problem with multicomputers is the difficulty in effectively programming them. Programmers are often faced with the choice of using high level programming tools that are easy to use but provide poor performance or low level tools that take advantage of specific hardware characteristics to obtain better performance but are difficult to use. In general, existing parallel programming environments do not provide any guarantee of performance and they provide little support for performance evaluation and tuning.

This dissertation explores an approach in which users are provided with programming support based on parallel programming paradigms. We have studied two commonly used parallel programming paradigms: Processor Farm and Divide-and-Conquer. Two runtime systems, *Pfarm* and *TrEK*, were designed and developed for applications that fit these paradigms. These systems hide the underlying complexities of multicomputers from the users, and are easy-to-use and topology independent. Performance models are derived for these systems, taking into account the computation and communication characteristics of the applications in addition to the characteristics of the hardware and software system. The models were experimentally validated on a large transputer-based system. The models are accurate and proved useful for performance prediction and tuning.

Pfarm and *TrEK* were integrated into *Parsec*, a programming environment that supports program development and execution tools such as a graphical interface, mapper, loader and debugger. They have also been used to develop several image processing and numerical analysis applications.

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, I would like to thank both my supervisors Dr. Samuel Chanson and Dr. Alan Wagner for their guidance, support and encouragement throughout my thesis research. I was very fortunate to have Dr. Wagner as my supervisor, whose direction and continuous involvement made it possible for my thesis research to take this final shape. I thank the members of my supervisory committee Dr. Mabo Ito and Dr. James Little for their valuable input to my research.

I thank Mandeep Dhami, David Feldcamp, Norm Goldstein, Kunhua Lin, Sameer Mulye and other past members of the parallel computation group for their cooperation during this research. I appreciate all the help provided by the system and administrative staff during my stay in the department. In addition, I would like to thank Chris Healey for proofreading parts of this thesis.

My thanks go to many fellow graduate students (Don Acton, Ian Cavers, Parag Jain, Sree Rajan, Vishwa Ranjan to name a few) for their friendship, help and support. Many thanks are also extended to numerous friends in UBC and outside who made our stay in Vancouver enjoyable.

I would like to acknowledge the financial support provided by the Canadian Commonwealth Scholarship and Fellowship Administration. I also thank the Government of India for sponsoring me for the fellowship.

Finally, I thank my wife Latha for her love, endless support and patience during the last five years.

Chapter 1

Introduction

Parallel processing is becoming popular with the advent of inexpensive, powerful microprocessors made possible by advances in VLSI technology. Several kinds of parallel computer architectures [Dun90] have been proposed and built. These parallel computers have been used successfully to achieve remarkable performance for applications in several areas including scientific computing, and signal and image processing. The domain of parallel computer architectures includes Single Instruction Multiple Data (SIMD) machines, and Shared memory and Distributed memory Multiple Instruction Multiple Data (MIMD) machines [Fly72].

Distributed memory MIMD machines, generally known as multicomputers [AS88], consist of a number of processors each with their own local memory, connected by a message-passing network. Several research and commercial multicomputers such as Hypercubes[Sei85], Transputer-based systems[Lim88] and iWARP[K+90] have been available since the mid 1980s. Multicomputer architectures have several advantages. These machines are able to take advantage of the latest and fastest microprocessor technology making them cost-effective in comparison to other parallel architectures. They are easily scalable compared to other architectures. In the case of reconfigurable machines, it is possible to take advantage of the communication patterns of the problem to improve performance. With these advantages, multicomputers are gaining importance as general purpose parallel machines useful for applications in a wide range of areas [FJL+88]. The focus of this research is on multicomputers and their effective use.

1.1 Motivation

The major stumbling block to the widespread use of multicomputers is the tremendous difficulty in effectively programming them. Software development for multicomputers has not kept pace with the advances in hardware technology [Kar87, KT88, CK91]. Message passing gives finer control over resources, but at the cost of added complexity. Users must address difficult problems such as partitioning, mapping, communication, synchronization, data distribution, and load balancing.

There are a few high level languages based on functional and logic programming models [Kog85, Dav85, FT90]. Those that are available provide high level abstractions with universal interfaces to all applications, but their overall performance is generally poor because of the difficulties in taking full advantage of the underlying structure of the application and the architecture. Most of the recent work on parallelizing compilers [PW86, CK88, HKT91] has focussed on extracting loop level and lower levels of parallelism. They are restricted to exploiting parallelism in certain loop structures and thus can improve performance only for certain problems such as SPMD (single program, multiple data) type programs that are data-parallel. Most of the commercial multicomputers provide low level machine-dependent environments [Lim88, lim87, Inc91] that can be used to achieve high performance. These environments provide very low level programming abstractions that makes program design a complex process. This leads to higher software development costs and programs that are not easily ported to other machines.

Difficulties in parallel programming do not end with the design and development of a working parallel program. The primary motivation for using parallel computers is to obtain higher performance for application programs. In general, existing programming environments do not provide any guarantee of performance, moreover they provide little support for performance evaluation and tuning. In the case of parallel systems, performance depends on the computation and communication characteristics of a parallel program in addition to the characteristics of the hardware and software system. Users generally have very little knowledge about the performance of their programs until they

are implemented and run. Even though one may think that using more processors will improve performance, this is not always the case. Simple models [Sto88] have shown that using more than a certain number of processors for a given application may not improve performance. In practice, it may actually degrade performance.

1.2 Methodology

Providing abstractions that are efficient and easy-to-use for programming multicomputers is a difficult problem. One recent approach to reconciling ease of use and reuse with performance is the construction of software components (libraries, modules, templates, skeletons) based on the parallel programming paradigms that have appeared in the literature [K+87, Nel87, Col89, Pri90]. These paradigms, taken together, represent the state of the art of parallel programming. Software components based on these paradigms can hide the complex distributed system code needed to implement the paradigm, thereby allowing the application programmer to concentrate on the computationally intensive code rather than parallelization and the coordination of the processors. Several projects such as Chameleon [Alv90], PIE [RSV90] and VMPP [Gab90] have looked at providing programming support for some of these paradigms on shared-memory machines.

It is difficult to obtain a single performance model that can be used for all applications on a parallel system. Performance depends on the computation and communication characteristics of the algorithm, in addition to the characteristics of the hardware and software system. There are some performance metrics, such as Amdahl's serial fraction [Amd67], experimentally determined serial fraction [KF90] and average parallelism [EZL89], which are based on simple characterizations of parallel systems. Although, they can be used to obtain rough bounds on performance, it is not as easy to use them for performance prediction and tuning. One approach to obtaining more accurate models that could be used for prediction and tuning is to model simpler and more restricted systems. Parallel programming paradigms are more restricted and sufficiently general to be of more general use. Models based on paradigms can take into account the computation and communication characteristics of the applications and also the characteristics of the hardware and software system.

Abstraction and added functionality that diminishes performance leads to constant re-design and re-implementation of the software component. Therefore, it is necessary to formalize these paradigms to better understand their expressiveness, their limitations, and most importantly their performance characteristics. It is important to understand the effect of scaling the component to execute on a larger number of processors. It must be possible to easily modify the behavior and performance characteristics of the component in order to take advantage of application specific optimizations (e.g., fixed versus variable data packets). By understanding the behavior and performance characteristics of the paradigm, it may be possible to guarantee performance and provide guidance to the use and design of these paradigms on different topologies or systems with different primitives. The challenge is to construct a system based on paradigms that is reusable and achieves close to optimal performance.

In this dissertation, we consider two task-oriented parallel programming paradigms, processor farm and divide-and-conquer. The processor farm paradigm is widely used in parallelizing applications in several areas [CHvdV88, BTU88, CU90, CCT93]. Divide-and-conquer is a well-known problem solving strategy in both sequential and parallel programming [AHU74, HZ83, GR88, Sto87].

The principal contributions of this dissertation research are:

- Development of performance models for two commonly used parallel programming paradigms: processor farm and divide-and-conquer.

  We have developed models that accurately describe the behavior and performance characteristics of processor farm and divide-and-conquer applications on multicomputers with the characteristics described in Section 3.2. These are realistic models that can help in understanding the capabilities and limitations of these paradigms. These models have been experimentally validated on a transputer-based system. They provide guidance for system design, and can be used for performance prediction and tuning.

- Design and development of execution kernels for processor farm and divide-and-conquer applications.

Execution kernels for both processor farm and divide-and-conquer have been designed and implemented on a transputer-based machine. The systems are topology independent, i.e., they can be used on machines of any size and topology. They have been integrated into a programming environment that includes supporting tools such as a graphical interface, mapper, loader and debugger. Several applications have been developed using these kernels.

1.3 Synopsis of the Dissertation

Chapter 2 provides an overview of the related literature that puts this dissertation work in context. It includes a discussion on various existing parallel programming approaches, highlighting their advantages and disadvantages, and comparing and contrasting our approach to these approaches. Existing performance measures and models for parallel systems are reviewed, emphasizing their applicability and limitations.

Chapter 3 describes the integrated approach we have taken to address the programming and performance modeling problems in multicomputers. This approach provides programming support based on parallel programming paradigms to the application programmers. We discuss the characteristics of processor farm and divide-and-conquer paradigms that are studied in this thesis. We describe how these paradigms can be used to parallelize several different applications.

In Chapter 4, we describe the design of *Pfarm*, a topology independent processor farm runtime kernel, detailing the trade-offs involved to make it efficient. *Pfarm* implements a distributed dynamic task scheduling strategy. The affect of process structure, scheduling, and buffering on performance has been investigated. We have developed a general analytical framework that can be used to derive performance models for processor farms on an arbitrary tree topology. For a fixed topology, we have shown that a breadth-first spanning tree provides maximum performance, and the steady-state performance of all breadth-first spanning trees are equal. Since a processor farm system behaves like a pipeline, we have also analyzed start-up and wind-down. The ideal architecture for *Pfarm* is a balanced k-ary tree, where k is the number of links on each processor.

Performance models for this case have been derived from the general framework. We also describe how the models can be used in performance tuning and restructuring of application programs.

The *Pfarm* system has been implemented on a 75 node transputer-based machine. The performance models were experimentally validated as reported in Chapter 5. The models are sufficiently accurate that they can be used to predict performance of this design on any tree topology. The robustness of the model under our assumption of average task size was tested for uniform and bimodal distributions. The model was accurate for the uniform distribution. In the case of the bimodal distribution, we found that the model remained accurate as long as the arrival pattern of the two task types was mixed.

In Chapter 6, we extend the design of *Pfarm* to provide runtime system support for divide-and-conquer applications. This system, called *TrEK* (Tree Execution Kernel), can execute divide-and-conquer computations of any degree and depth on an arbitrary tree topology. *TrEK* is designed to make use of intermediate processors for subtask processing in order to increase the overall performance. We expanded the general analytical framework given for *Pfarm* to derive performance models for fixed degree divide-and-conquer applications on an arbitrary tree topology. Experimentally, we found that performance depends on the depth and number of leaves in the tree topology. Thus, on a fixed topology, a breadth-first spanning tree with a maximum number of leaves achieves maximum performance. With a reconfigurable network, a g-ary balanced tree, where g is the number of links on each node, provides maximum performance. We derived models that can predict the performance of any fixed k-ary divide-and-conquer computation on any g-ary balanced tree topology.

Chapter 7 describes the experiments conducted to validate the performance models for divide-and-conquer. The experiments show that our framework performs well even for applications that consist of a single large divide-and-conquer task in addition to those with a flow of tasks. In some cases, it is possible to use the processor farm strategy for divide-and-conquer applications. We found that *TrEK* outperformed *Pfarm* for applications with larger tasks and for those applications that consist of a smaller

number of tasks.

In order to make it easier for application programmers to use these paradigms on a multicomputer system, a programming environment that supports all phases of program development and execution is needed. In Chapter 8, we describe *Parsec*, an on-going project at the University of British Columbia in developing an integrated programming environment for the support of paradigms. *Parsec* provides *Pfarm* and *TrEK* with supporting tools such as a graphical interface, mapper, loader, monitor and debugger. We have also discussed applications that have been developed using *Pfarm* and *TrEK*. Chapter 9 provides a summary of the dissertation with a discussion of topics for future research.

Chapter 2

Background and Related Work

The relative ease with which it is possible to build inexpensive, high-performance multicomputers using commodity microprocessors has made multicomputers very popular. The major problem with multicomputers is the difficulty in effectively programming them. Programmers must either use a high level programming tool that is easy to use but provides poor performance or a machine dependent low level tool that can provide high performance but is difficult to use.

To be successful, a parallel programming environment should address both the basic issues, namely programming and performance. First, programmers should be provided with easy-to-use programming abstractions that hide the complexities of the system. Second, the environment should be able to assist programmers in obtaining the maximum performance on a given parallel architecture for their applications.

In this chapter, we provide a overview of the related literature. Section 2.1 describes various parallel programming approaches, emphasizing their advantages and disadvantages. In Section 2.2, a review of the literature on existing performance measures and models for parallel computing is provided.

2.1 Parallel Programming Approaches

In this section, various parallel programming approaches are reviewed, highlighting their advantages and disadvantages, and comparing and contrasting them with our approach.

We also review the existing research on identifying parallel programming paradigms.

2.1.1 High Level Approaches

Parallelizing Compilers

Parallelizing compilers are aimed at extracting loop level and lower levels of parallelism in a sequential program. Considerable research work is being done in developing compilers that automatically parallelize FORTRAN DO loops [PW86, CK88, HKT91]. The programmers write sequential programs in standard FORTRAN, and the compiler analyzes data dependencies and uses parallelizing and vectorizing constructs to optimize the program for a given parallel hardware.

With automatic parallelizing compilers, users need not be concerned with writing explicitly parallel code. In some cases, users can provide compiler directives for program partitioning and mapping. Compilers generally perform local optimizations which may not always lead to an overall improvement in performance. They have to use conservative values for data unknown at compile time. Parallelizing compilers have been successfully used on multicomputers for certain classes of problems such as SPMD (Single Program Multiple Data) programs.

In comparison, we have studied task-oriented paradigms on multicomputers. Our approach concentrates on global optimizations that lead to overall improvement in performance of application programs. This is done by considering classes of applications separately, and identifying their characteristics to decide on the necessary global optimizations to efficiently run them on a particular hardware system.

High-Level Languages

There is a group of researchers that advocate the use of high-level languages based on functional, logical and data-flow models of computation [Kog85, Dav85, Den80]. Using these languages, the programmer needs only to write a high-level declarative description of the algorithm, which is free of concurrency. The compiler and the runtime system produce code suitable for a specific parallel system.

The advantage of being able to write programs in a very high level is generally outweighed by the resulting poor performance. It is very unlikely that the standard implementation decisions used by the compiler will be optimal for all situations. Programmers who understand the specific structure of their algorithms can always do better optimizations than the generalized transformations included in a compiler. In our approach, different classes of applications are considered separately, and good optimizations for each of them are obtained by understanding the structure of the underlying algorithms.

2.1.2 Low level Approaches

In multicomputers such as the transputer-based [lim87] and C40-based [Inc91, Inc92] machines, the user is totally responsible for implementing parallelism. The programming environments provided in these cases consist of languages such as occam [Lim84] or extended C that provide process creation and inter-process message passing. The programmer is responsible for partitioning the work into processes and mapping them to exploit the parallelism provided by the hardware. The programmer also has to manage communication between processes.

It is possible to extract maximum performance out of the system if the programmer has good knowledge of the underlying hardware architecture and how well it can be used for a given application program. Even though high performance is achievable, it is difficult since the programming environments provide minimal support for these machines.

In our approach, the programming system provided to the users efficiently implements parallelism on a given machine and manages communication among processors. The user has to concentrate only on the application dependent sequential code.

2.1.3 Other Approaches

The Linda project advocates the use of a coordination language such as Linda in conjunction with computational languages like C or FORTRAN to make parallel programming easier [GC92]. A coordination language provides operations to create multiple execution threads (or processes) and supports communication among them. Linda[CG89, ACG91]

consists of a few simple *tuple-space* operations. Adding these tuple-space operations to a computational language produces a parallel programming dialect such as C-Linda. In this case, the programmers are responsible for creating and coordinating multiple execution threads. Linda provides a shared memory paradigm independent of the underlying parallel architecture. The processes can communicate and synchronize using the tuple-space, which is in fact a shared memory.

Implementation of the Linda tuple-space on a distributed memory machine is generally difficult since the tuple space has to be distributed and replicated, which can lead to poor performance. With our approach, the system takes responsibility for process creation and coordination, rather than the user.

Foster and Overbeek[FO90] propose an approach called *bilingual programming.* In this approach, the key idea is to construct the upper levels of an application in a high-level language, while coding the selected low-level components in low-level languages. They argue that this approach permits the advantages of the high-level notation (expressiveness, elegance, conciseness) to be obtained without the usual cost in performance. They introduce a particular bilingual approach in which the concurrent programming language Strand [FT90] is used as the high-level language and C or Fortran is used to code low-level routines. Strand provides a high level notation for expressing concurrent computations.

With this approach, overall performance is determined by the decisions on how to partition concurrent processes into tasks and map them onto various nodes. The user is responsible for partitioning and mapping, although there are some tools which can provide guidance. Our runtime systems take care of creating concurrent processes and communicating among them. Each system efficiently implements partitioning and scheduling for a particular class of applications. The performance models can be used for restructuring application programs to obtain better performance.

2.1.4 Parallel Programming Paradigms

There are several well-known programming paradigms such as divide-and-conquer, branch-and-bound and dynamic programming techniques that are commonly used in

designing sequential algorithms. These paradigms are not exactly algorithms, but they are problem solving techniques or high level methodologies that are common to many efficient algorithms.

We can find similar problem solving techniques that are commonly being used in designing parallel algorithms. Identifying and analyzing useful parallel programming paradigms will help the programmer in understanding parallel computation and in the difficult process of designing and developing efficient parallel algorithms.

In general, programming paradigms encapsulate data reference patterns. In the case of parallel programming paradigms, they encapsulate underlying communication patterns. Since they identify useful communication patterns, they can help in designing architectures that can effectively support commonly used communication patterns. The analysis of these paradigms can provide guidelines for designing programming tools that can assist application programmers in obtaining better performance on a given parallel machine.

The following paragraphs summarize the related research on identifying and understanding useful parallel programming paradigms.

In 1989, Kung et. al. [Kun89] identified several computational models based on their experiences in parallel algorithm design and parallel architecture development. These models characterize the interprocessor communication and correspond to different ways in which cells in 1D processor arrays exchange their intermediate results during computation. The models are:

1. Local computation
2. Domain partition
3. Pipeline
4. Multi-function pipeline
5. Ring
6. Recursive computation
7. Divide-and-conquer
8. Query processing
9. Task Queue

Fox [FJL+88] and Karp [Kar87] have discussed SPMD paradigm for programming shared and distributed memory multicomputers. In the SPMD model, the same program is executed on all the processors. Processors communicate their intermediate results to

their neighbors and synchronize at a barrier point. Fox and others[FJL+88] have successfully used the SPMD model to solve a number of large applications on multicomputers.

Nelson[Nel87] has studied compute-aggregate-broadcast, divide-and-conquer, pipelining and reduction paradigms for distributed memory parallel computers. He has discussed how these paradigms can be used to develop algorithms for solving many numerical and non-numerical applications. He has also studied the contraction problem, the problem arising when an algorithm requires more processors than are available on a machine, for algorithms based on these paradigms.

Cole[Col89] advocates an approach in which the users are presented with a selection of "Algorithmic Skeletons" instead of an universal programming interface. Each skeleton captures the essential structure of some particular problem solving style or technique. To solve a particular problem, the user is required to select a skeleton which describes a suitable problem solving method. The procedures and data structures are added to the skeleton to customize it to the specific problem. Since each instance of these procedures will be executed sequentially, they can be specified in any sequential programming language. He has discussed four different skeletons - Divide-and-conquer, Task Queue, Iterative Combination and Cluster. He proposes to embed suitable topologies for various skeletons on a grid architecture. In terms of performance, he has focussed on the asymptotic efficiency with which a large grid of processors can implement a system with respect to the performance of a single processor.

Pritchard and Hey [Pri90, Hey90] discuss three useful paradigms for programming transputer arrays: Processor Farm, Geometric Array and Algorithmic Pipes. Processor Farm uses a manager-workers setup to solve an application that consists of a large number of independent tasks. Geometric Array is same as the SPMD model mentioned earlier and Algorithmic Pipes is similar to the pipeline approach.

PIE project [RSV90] uses parallel programming paradigms as an intermediate layer of abstraction, called implementation machine (IM) level, between the application level and the physical machine level for uniform memory access multiprocessors. Each IM has two representations: an analytical representation and a pragmatic representation. The analytical representation helps in predicting the performance of a class of applications

using the IM. The model predicts the upper bound and lower bound on performance of an application that uses this IM. A pragmatic representation of IMs is made available in the form of modifiable templates. All necessary communication and synchronization for the IM are correctly and efficiently implemented in the template. All the user needs to do is to insert the application dependent code. With the help of the IM layer, the user can write performance efficient parallel programs with relative ease. The analytical models help the user to select the most appropriate or efficient IM for a given application and parallel machine. Two IMs, master-slave and pipeline, have been implemented on a Encore Multimax, a bus-based shared memory multiprocessor.

In this thesis, we explore an approach in which users are provided with programming support based on parallel programming paradigms for multicomputers. This approach is similar to Cole's proposal of Algorithmic Skeletons. In contrast to Cole's theoretical study of how various skeletons can be implemented on a grid architecture, we have implemented runtime systems for two widely used paradigms, Processor Farm and Divide-and-Conquer, that are topology independent. Performance models derived in this thesis are analytical models, unlike Cole's asymptotic models, and hence can be used in performance tuning. Our approach is similar to that followed by PIE. It differs in the underlying parallel architectures being considered and the apparent fact that we can obtain accurate models on multicomputers.

2.2 Performance Modeling

In the case of sequential computation, performance can be adequately characterized by the instruction rate of the single processor and the execution time requirement of the software on a processor of unit rate. Predicting the performance of a parallel algorithm on a parallel architecture is more complex. Performance depends on the computation and communication characteristics of the algorithm, in addition to the characteristics of the hardware and software system.

In order to use parallel systems effectively, it is important to understand the performance of parallel algorithms on parallel architectures. This can help in determining

the most suitable architecture for a given algorithm. It can also help in predicting the maximum performance gain which can be achieved. In this section, we summarize the relevant research in understanding the performance behavior of parallel systems and highlight the applicability and limitations of each.

2.2.1 Performance Measures and Models

It is a well known fact that the speedup for a fixed-size problem on a given parallel architecture does not continue to increase with an increasing number of processors, but tends to saturate or peak at a certain value.

In 1967, Amdahl [Amd67] argued that if s is the serial fraction in an algorithm, then the speedup obtainable is bounded by $1/s$ even when an infinite number of processors are used. For an N processor system, speedup is given by

$$\frac{1}{s + (1-s)/N}.$$

This observation, which is generally known as *Amdahl's Law*, has been used to argue against the viability of massively parallel systems.

In the recent years, researchers have realized that it is possible to obtain near-linear speedup by executing large problems. In 1988, Gustafson and others at Sandia National Lab [Gus88] were able to obtain near-linear speedup on a 1024-processor system by scaling up the problem size. Gustafson argues that in practice, users increase the problem size when a powerful processor is made available; hence, it may be more realistic to assume *runtime* as constant instead of *problem size*. He introduced a new measure called *scaled speedup*, defined as the speedup that can be achieved when the problem size is increased linearly with the number of processors. For an N processor system, *scaled speedup* is given by $N + (1 - N) \times s$.

Karp and Flatt[KF90] have used experimentally determined serial fraction as a metric in tuning performance. The experimentally determined serial fraction, f is defined as

$$\frac{1/S - 1/N}{1 - 1/N},$$

where S is the speedup obtained on an N-processor system. If the loss in speedup is only due to the serial component, that is, there are no other overheads, the value of f is

exactly equal to the serial fraction s used in Amdahl's law. With the help of experimental results, they argue that this measure provides more information about the performance of a parallel system. If f increases with N, then it is considered an indicator of rising communication and synchronization overheads. An irregular change in f as N increases would indicate load balancing problems.

Eager, Zahorajan and Lazowska[EZL89] use a simple measure called *average parallelism* to characterize the behavior of a parallel software system. The software system is represented by an acyclic directed graph of subtasks with precedence constraints among them. Average parallelism is defined as the average number of processors that are busy during the execution time of the software system, given an unbounded number of available processors. Once the average parallelism A is determined, either analytically or experimentally, the lower bounds on speedup and efficiency are given by

$$\frac{NA}{(N+A-1)} \qquad \text{and} \qquad \frac{A}{(N+A-1)}$$

respectively. This measure can be used only if the parallel system does not incur any communication overheads or whenever these overheads can be easily included as part of the tasks.

Kumar and Rao[KR87] have developed a scalability measure called the *isoefficiency function*, which relates the problem size to the number of processors necessary for an increase in speedup proportional to the number of processors used. When a parallel system is used to solve a fixed-size problem, the efficiency starts decreasing with an increase in the number of processors as the overheads increase. For many parallel systems, for a fixed number of processors, if the problem size is increased then the efficiency increases because the overhead grows more slowly than the problem size. For these parallel systems, it is possible to maintain efficiency at a desired value (between 0 and 1) for an increasing number of processors, provided the problem size is also increased. These systems are considered to be scalable parallel systems. For a given parallel algorithm, for different parallel architectures, the problem size may have to increase at different rates with respect to the number of processors in order to maintain a fixed efficiency. The rate at which the problem size is required to grow with respect to the number of processors to keep the efficiency fixed essentially determines the degree of scalability of

the parallel algorithm for a specific architecture. If the problem size needs to grow as $f_E(p)$ to maintain an efficiency E, then $f_E(p)$ is defined as the *isoefficiency function* for efficiency E.

If the problem size is required to grow exponentially with respect to the number of processors, then the algorithm-architecture combination is poorly scalable since it needs enormously large problems to obtain good speedups for a larger number of processors. On the other hand, if the problem size needs to grow only linearly with respect to the number of processors, then the algorithm-architecture combination is highly scalable. Isoefficiency analysis has been used in characterizing the scalability of a variety of parallel algorithm-architecture combinations [GK92]. Using isoefficiency analysis, one can predict the performance of a parallel program on a larger number of processors after testing the performance on a smaller number of processors.

Stone[Sto88] has used a simple model to determine how granularity affects the speedup on a multiprocessor. The model considers an application program that consists of M tasks and obtains the maximum speed with which this program can be executed on an N processor system. It assumes that each task executes in T_p units of time. Each task communicates with every other task at an overhead cost of T_c units of time when the communicating tasks are not on the same processor, and at no cost when the communicating tasks are on the same processor. The results of this model indicate that the speedup is proportional to N up to a certain point. After this, as N increases, the speedup approaches a constant asymptote which can be expressed as a function of the task granularity. This model gives a general picture of how granularity and overhead affect the performance of a multiprocessor system. It also gives some indication of the importance of minimizing overhead and selecting a suitable granularity. Stone's studies indicate that there is some maximum number of processors that is cost-effective, and this number depends largely on the architecture of the machine, on the underlying communication technology, and on the characteristics of each specific application.

Flatt and Kennedy[FK89] have derived some upper bounds on the performance of parallel systems taking into account the effect of synchronization and communication overheads. They show that if the overhead function satisfies certain assumptions, then

there exists a unique value N_o of the number of processors for which the total execution time for a given problem size is minimum. However, for this value, the efficiency of the system is poor. Hence they recommend that N should be chosen to maximize the product of speedup and efficiency and analytically compute the optimal values of N. A major assumption in their analysis is that the per-processor overhead grows faster than $\mathcal{O}(N)$, which limits the applicability of the analysis.

Performance metrics such as serial fraction (s and f) and average parallelism (A) are simple measures that can be used to obtain rough bounds on performance. These cannot be easily used for performance prediction and tuning, especially for multicomputers in part because they neglect communication overheads. Also, the values of these parameters often cannot be obtained easily. We have concentrated on considering the characteristics of both the system and the applications in order to obtain accurate performance models that can be used for performance prediction and tuning. Our models take into account all the communication overheads involved in implementing different classes of applications on multicomputers. The values of the parameters used in our models can be determined in a relatively easy manner, and we discuss some of the techniques for obtaining them.

2.2.2 Integration

There has been very little work done in integrating performance tuning into programming environments to provide performance-efficient parallel programming. To make best use of the underlying parallel architecture for an application program, in addition to programming support, users must be provided with performance models that can help in predicting how well their programs are going to perform. The environment should be able to assist the programmers in restructuring their applications to improve performance.

PIE[SR85] addresses the issues of integrating performance tuning into the programming environment through the support of specific implementation paradigms coupled with a performance prediction model. The model provides performance trade-off information for parallel process decomposition, communication, and data partitioning in the context of a specific implementation paradigm and a specific parallel architecture.

Gannon[AG89] describes an interactive performance prediction tool that can be used by the user to predict execution times for different sections of a program. This performance predictor analyzes FORTRAN programs parallelized by an automatic parallelizing and vectorizing compiler targeted for the Alliant FX/8. Programmers can use the predictor to estimate the execution time for a segment of the code produced by the compiler. The predictor uses a database to estimate the total number of CPU cycles needed for the segment. They also incorporate a simple model of memory contention into the predictor to include the effects of caching. This predictor can be used only to predict the execution time of a segment of the program; it does not give any specific information about the overall performance of a parallel program.

Kennedy and Fox[BFKK91] have worked on an experimental performance estimator for statically evaluating the relative efficiency of different data partitioning schemes for any given program on any given distributed memory multiprocessor. The performance estimator is aimed at predicting the performance of a program with given communication calls under a given data partitioning scheme. This system is not based on a performance model. Instead, it employs the notion of a training set of kernel routines that test various primitive computational operations and communication patterns on the target machine. The results are used to train the performance estimator for that machine. This training set procedure needs to be done only once for each target machine, during the environment or compiler installation time. Although the use of a training set simplifies the task of performance estimation significantly, its complexity lies in the design of the training set program, which must be able to generate a variety of computation and data movement patterns to extract the effect of the hardware/software characteristics of the target machine on the performance. The authors argue that real applications rarely show random data movement patterns and there is often an inherent regularity in their behavior. They believe that their training set program will probably give fairly accurate estimates for a large number of real applications, even though it tests only a small (regular) subset of all the possible communication patterns.

We have integrated the programming and performance tuning support into *Parsec* (described in Chapter 8). *Parsec* is an on-going project at the University of British

Columbia which is aimed at developing an integrated programming environment to support several parallel programming paradigms. It includes supporting tools such as a graphical interface, mapper, loader and debugger. The programmers can make use of performance models to predict the performance of their applications, and also can obtain optimal values for system parameters such as the number of nodes and topology that can lead to maximum performance. The environment allows programmers to easily change the parameters and accordingly does the necessary mapping and loading. Some of the techniques that can be used to determine the values of the application dependent parameters are described in Chapter 8.

Chapter 3

Methodology

The diversity of parallel computing architectures and their underlying computation models makes it particularly difficult to find universal techniques for developing efficient parallel programs. Choosing an appropriate parallel machine for a given application is a difficult process. Furthermore, in most of the existing programming environments available on multicomputers, the user is responsible for managing both parallelism and communication. As explained in Chapter 2, identifying and analyzing useful parallel programming paradigms may help programmers in the difficult process of developing efficient parallel algorithms. In this chapter, we present an approach based on parallel programming paradigms for developing efficient programs for multicomputers.

Section 3.1 describes an integrated approach we have taken to address the programming and performance modeling problems on multicomputers. In Section 3.2, we explain the multicomputer system model used in this dissertation. Section 3.3 describes the transputer-based multicomputer system that is used as an experimental testbed in this research. This approach has been used to develop programming support and performance models for two commonly used parallel programming paradigms, processor farm and divide-and-conquer. In Section 3.4, we discuss the characteristics of these two task-oriented programming paradigms and how these paradigms can be used for various kinds of applications.

3.1 An Integrated Approach

This approach provides application programmers with abstractions based on commonly used parallel programming paradigms. The application programmers are provided with a set of Virtual Machines (VMs), where each virtual machine corresponds to a parallel programming paradigm. Each virtual machine consists of an analytical performance model, and an efficient runtime system that can be used to run applications that fit into the corresponding paradigm. The user has to choose one of the virtual machines that corresponds to the paradigm that can be used to solve his application problem.

With this approach, the users are not responsible for implementing parallelism and communication. Each runtime system implements parallelism and all the necessary communication and synchronization needed for running the corresponding class of applications in an efficient manner. It also implements other system dependent aspects such as task scheduling and load balancing. Such a runtime system can be efficiently implemented by a systems programmer who understands the complexities of the underlying hardware and software system. The runtime system provides a simple interface to the user. The user has to write only the application dependent code and execute it with the runtime system. This approach eliminates the difficulties in programming multicomputers and reduces the software development cost. Also, as the user code does not contain any system dependent parts, it is portable across different machines and systems on which the virtual machine implementations are available.

The analytical performance model helps in predicting the performance of application programs that use the particular virtual machine. Some of the parameters of the model are application program dependent and the others are dependent on the characteristics of the underlying physical machine and software system. The values of the system dependent parameters can be estimated once the runtime systems are implemented. The application dependent parameters are either estimated or measured (either from a serial program or from a scaled down parallel program). Once these parameter values are known, the model can be used to predict the actual performance of an application program on a given parallel system. It can also be used in performance tuning, either to

choose the optimal number of nodes to be used for a given application or to restructure the application to maximize its performance.

Two virtual machines corresponding to commonly used parallel programming paradigms, processor farm and divide-and-conquer, were developed. In Section 3.4, we describe the characteristics of these two task-oriented paradigms.

3.2 System Model

In this thesis, we consider distributed memory parallel computers (multicomputers). The system consists of processor nodes connected by an interconnection network such as a chain, tree, mesh, hypercube etc. with the underlying support for point-to-point communication. Each processor node consists of a CPU with its own local memory and hardware support for communication links.

Execution kernel designs are based on the following assumptions on the characteristics of the underlying system. The kernels assume a reliable point-to-point communication mechanism. Furthermore, it is assumed that the data can be transferred simultaneously on all the links and that data transfer can be overlapped with the computation. However, for each message, there is a CPU start-up cost that cannot be overlapped with the computation. This time may include hardware set-up costs, context switch times, and other system software overheads. Message start-up is an important overhead that significantly affects performance. In addition, it is assumed that the system supports concurrent processes on each processor with process scheduling and levels of priorities. This support could be available either in hardware (as in Transputers) or in software (as in TI C40s).

Performance models are developed assuming homogeneous processors and links. A linear cost communication model is assumed (i.e., for every message, there is a CPU cost for starting the communication, and a transfer cost proportional to the message size). The time for a processor node to send a message of length m to its neighbor is given by $\beta + \tau m$, where β is the CPU start-up cost described in the previous paragraph and τ is the transfer rate of the communication links.

3.3 Experimental Testbed

In this section, we describe the transputer-based multicomputer in the Department of Computer Science at the University of British Columbia that is used as an experimental testbed in this thesis. Performance models derived in Chapter 4 and 6 for processor farm and divide-and-conquer are applicable for any multicomputer system that satisfies the system model described in Section 3.2. In addition, the corresponding runtime system designs can be implemented on any similar multicomputer system.

3.3.1 Hardware System

The transputer-based multicomputer consists of 75 T800 transputer nodes and 10 crossbar switches. The system is hosted by a Sun-4 workstation via VME bus, with 4 ports that connect the system to the host. There are 64 nodes with 1 MB of external memory and 10 nodes with 2 MB. There is a special node that has 16 MB, and is used as the manager node for both *Pfarm* and *TrEK*.

The INMOS T800 transputer is a 32 bit microprocessor with a 64 bit floating point unit on chip. A T800 running at 20MHz has a sustained processing rate of 10 MIPS and 1.5 Mflops. It has 4 KB of on-chip RAM and four bit-serial communication links. These communication links allow networks of transputer nodes to be constructed by direct point to point connections with no external logic. Each link runs at an operating speed of 20 Mbits/sec and can transfer data bidirectionally at up to 2.35 MB/sec.

The T800 processor has a microcoded scheduler that enables any number of concurrent processes to be executed together, sharing the processor time. It supports two levels of priority for the processes: high and low. A high priority process, once selected, runs until it has to wait for a communication, a timer input or until completion. If no process with a high priority is ready to proceed, then one of the ready low priority processes is selected. Low priority processes are time sliced every millisecond. A low priority process is only permitted to run for a maximum of two time slices before the processor deschedules it at the next descheduling point. Process context switch times are less than 1 μs, as little state needs to be saved. The transputer has two 32 bit timer clocks. One timer

is accessible only to high priority processes and is incremented every microsecond, and the other is accessible only to low priority processes and is incremented every 64 μs.

Each C004 crossbar switch has 32 data links and a control link. These are programmable through the control link and each can have 16 bidirectional connections. Thus, the system is reconfigurable and an appropriate interconnection network for an application can be chosen. As the system is not fully connected, there are some restrictions on the possible configurations. For *Pfarm* and *TrEK*, we statically configure the system into an appropriate interconnection network.

3.3.2 Software Environments

Pfarm and *TrEK* have been implemented using C on two different software environments that are available on our multicomputer: Logical Systems and Trollius. The Logical Systems environment [Moc88, Sys90] includes a library of process creation and communication functions. The environment has an utility called *ld-net* that runs on the host and downloads the executable programs onto a network of transputers.

In the Trollius environment [BBFB89, F$^+$90], the programs are run on top of a kernel on each transputer node. The Trollius kernel manages and synchronizes any number of local processes. There are two levels of message passing in Trollius. The kernel level allows communication between processes on the same node. The network level allows communication between processes on different nodes, as well as between processes on the same node. There are four sub-levels of message passing within the Trollius network level, representing different functionality/overhead compromises. They are, in order of increasing functionality and overhead, the physical, datalink, network and transport sub-levels. Trollius provides both blocking and non-blocking communication functions. It also includes a set of utility programs that run on the host, which can be used to load, monitor and control the programs on transputer networks.

3.4 Task-oriented Paradigms

In this section, we describe the characteristics of two task-oriented parallel programming paradigms, processor farm and divide-and-conquer, that are considered in this thesis.

3.4.1 Processor Farm

Processor farm is one of the most widely used strategies for parallelizing applications. There are a large number of scientific and engineering applications that consist of repeated execution of the same code, with different initial data. In addition, there is little or no dependency among the different executions of this code. These different executions can be considered as a set of tasks and can be executed in parallel. When we use multiple processors to execute these tasks in parallel, even though all the processors are executing the same code, they may not be executing the same instruction at any given time as they execute different parts of the code depending on the initial data. Therefore, it is not possible to use SIMD (Single Instruction Multiple Data) parallel machines to run these applications. These kind of applications can be run efficiently on MIMD (Multiple Instruction Multiple Data) parallel computers. Even though shared memory MIMD machines can be used for this class of applications [Alv90], distributed memory MIMD machines are being widely used because they are scalable and cost-effective. Processor farms are described in many transputer programming books [Lim88, Gal90, Cok91]. This strategy has been used in parallelizing applications in several areas [CHvdV88, BTU88, CU90, SS91, CCT93, NRFF93].

The typical setup to run these applications is to use a "farm" of worker processors that receive tasks from, and return results to, a manager processor. Each worker processor runs the same program and depending on the data received for individual tasks, it may execute the same or different parts of the program. No communication is required among the worker processors to execute the tasks. The manager processor has to communicate with the worker processors to distribute the tasks and to receive the results.

The ideal architecture for this "manager-workers" setup is one in which each worker

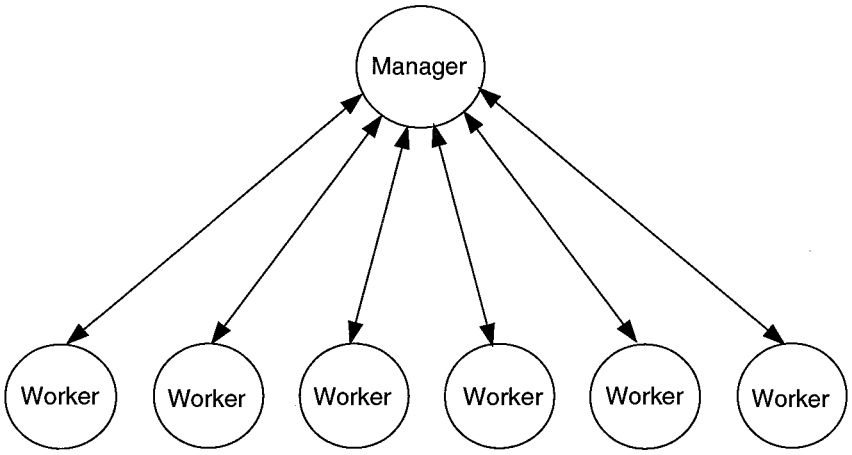


Figure 3.1: Ideal Manager-Workers Architecture

processor is directly connected to the manager processor as shown in Figure 3.1. In practice, this can not be realized as the processor nodes in distributed memory machines generally have a fixed degree of connectivity (e.g., in the transputer case, each processor has four links and a network of transputers can be configured using crossbar links). It is possible to use crossbar switches to dynamically reconfigure the connection between the manager and the worker processors such that every worker will be connected to the manager directly for a certain period of time [Hom93]. The need for special hardware in addition to not being scalable makes this configuration less useful. Also, many commercial machines use a fixed interconnection network such as mesh or hypercube for interconnecting their processor nodes. Thus, one has to use an interconnection network in implementing processor farm applications, and the topology has to be chosen with a view of minimizing the communication costs in distributing tasks and collecting results. Since tree architectures that have a minimum possible number of hops to each worker from the manager incur minimum communication costs, they are best suited for implementing processor farm applications.

There are several variants of processor farm that increase in applicability. In the generic description of a processor farm, the application program consists of a number of independent tasks that have to be executed on a network of worker processors. All the tasks may be of the same type in which case the workers execute the same code

or different parts of the code depending on the initial data. The computation time requirements might vary from task to task depending on their data. The manager might have all the tasks readily available to start with, or there could be some computation involved in producing these tasks. Also, in the case of real-time applications, the data for individual tasks arrive in real time.

Processor farms can also be used to execute applications that consist of multiple types of tasks. In this case, the worker processors have to be loaded with the code needed to execute each type of task with the appropriate code chosen, according to its type at runtime. Cramb and Upstill [CU90] discuss a processor farm implementation of such an application.

It is also possible to use processor farms to execute application programs that consist of multiple phases of computation [Son93]. In these applications, the tasks might have some dependency. After a phase of computation, a new set of tasks for the next phase are generated based on the results of the current phase. Also, some applications might consist of a number of distinct phases of computations out of which some phases could be executed efficiently using a processor farm [BTU88]. In this case, the processor farm acts as a computational server that is called at different points in the application program to execute a set of tasks.

Most of the processor farm designs discussed in the literature are overly simplistic and hide the issues that affect their performance and reuse. Poor reuse of code is a major problem in parallel programming and this remains the case in processor farm applications. There is very little work done in understanding how the various parameters such as the hardware topology, computation and communication requirements of the tasks affect overall performance.

In addition to making it reusable and topology independent, the *Pfarm* system described in Chapter 4 was designed considering all the factors that affect the performance. Performance models were derived from these realistic implementations of *Pfarm*. The models have been experimentally validated and are found to be accurate. Thus, they can be used in performance tuning to maximize performance.

3.4.2 Divide-and-Conquer

Divide-and-conquer is a well-known problem solving strategy used in deriving efficient algorithms for solving a wide variety of problems on sequential machines. Efficient divide-and-conquer algorithms have been used for solving problems in several areas such as graph theory, sorting, searching, computational geometry, Fourier transforms and matrix computations [AHU74, HU79, Sed83, Ben80]. It is also a useful strategy in hardware design as mentioned by Ullman in [Ull84].

The divide-and-conquer strategy can be briefly stated as follows: A large instance of a problem is divided into two or more smaller instances of the same problem. The results of smaller instances, called sub-problems, are combined to obtain the final result for the original problem. Sub-problems are recursively divided until they are indivisible and can be solved by a non-recursive method.

On uniprocessor systems, after dividing the original problem, sub-problems are solved sequentially. Sequential divide-and-conquer results in a tree structure of sub-problems. Several researchers have discussed the usefulness of the divide-and-conquer strategy in parallel processing [HZ83, GM85, GR88, Sto87]. On parallel systems, sub-problems can be solved concurrently provided that the system has sufficient parallelism. Problem splitting and combining of results can also make use of the available parallelism. These operations require interprocessor communication for distributing the data and for receiving the corresponding results. As the sub-problems are independent, there is no need for communication among the processors working on different sub-problems.

In its most general setting, a divide-and-conquer algorithm can be described as a dynamically growing tree structured computation, where initially there is a single problem and sub-problems are created as the problem is recursively divided. The number of subproblems and the depth of the tree may depend on the data and thus known only at runtime. For example, evaluation trees of functional and logic programs have the above characteristics. In the case of applications such as matrix multiplication and FFT, the degree of division and the depth of the computation tree is fixed. In some applications such as Quicksort, the problems are not equally divided.

There are real-time applications in areas such as vision and image processing, in which there is a continuous stream of real-time data and each data set has to be processed with a divide-and-conquer algorithm. Also, there are some non real-time applications in areas such as numerical analysis that consist of a set of problems, each of which can be solved using a divide-and-conquer algorithm.

In chapter 6, we describe the design of a runtime kernel called *TrEK*(Tree Execution Kernel) that provides runtime system support for divide-and-conquer applications on multicomputers. *TrEK* is designed such that it can execute divide-and-conquer computations of any degree and depth on an arbitrary tree topology. To improve the overall performance, *TrEK* makes use of the intermediate processors to process subproblems in addition to splitting and joining. The task-oriented framework chosen here assumes that there is a flow of divide-and-conquer tasks. This framework is well suited for applications that consist of a number of divide-and-conquer tasks. It can also be used for applications in which the problem consists of a single divide-and-conquer computation tree with large degree and depth. In this case, when we run such an application on a processor tree of smaller degree, the flow of tasks approximate the computation entering the subtrees. This framework allows us to derive performance models that accurately describe the behavior and performance characteristics of *TrEK*.

In the following paragraphs, we discuss how this work contrasts with the related work in using divide-and-conquer for parallel processing.

Horowitz and Zorat [HZ83] have outlined how appropriately designed multiprocessors whose logical structure reflect the tree structure of divide-and-conquer can be used to efficiently execute divide-and-conquer algorithms. They discuss the data movement problem in hardware configurations that have local memory, global memory via common bus and global memory augmented by local caches. Their proposal of a local memory architecture consists of processors with local memory connected as a full k-ary tree for executing k-ary divide-and-conquer problems. In contrast with their model, *TrEK* makes use of intermediate processors in addition to leaf processors for processing of subproblems to improve overall performance. Also, *TrEK* can execute divide-and-conquer problems of any degree and depth on distributed memory machines with any given topology.

Stout [Sto87] has discussed the usefulness of the divide-and-conquer strategy on distributed memory parallel computers for solving image processing problems. He has presented a divide-and-conquer algorithm for the connected components problem. He has analyzed some of the requirements of this problem, and outlined some of the implications for machine architectures and software. Nelson [Nel87] has studied how the divide-and-conquer paradigm can be used in designing parallel algorithms. He has discussed and presented parallel algorithms based on sequential divide-and-conquer algorithms for Batcher's Bitonic Sort, Matrix Multiplication, and Fast Fourier Transform(FFT) problems. Contraction of these algorithms on a binary n-cube show that different algorithms required different contractions to obtain good results. Our work focuses on developing efficient programming support for divide-and-conquer applications on multicomputers to hide the underlying complexities of the parallel machine from the programmer.

McBurney and Sleep [MS88] have done experimental work on implementing divide-and-conquer algorithms on a transputer-based system. Their ZAPP architecture is a virtual tree machine that is capable of dynamically mapping a process tree onto any fixed, strongly connected network of processors. Each processor runs a ZAPP kernel that implements the divide-and-conquer function. Each processor performs a sequential depth first traversal of the process tree, constructing sub-problems. Parallelism is introduced by allowing immediate neighbor processors to steal sub-problems. It is difficult to obtain any performance model for this framework. The task distribution strategy in *TrEK* differs from that used in ZAPP. Unlike ZAPP, *TrEK* does not allow a node to grab subtasks from its output queue for processing. This restriction allows us to model the system and does not degrade performance.

Cole, in his algorithmic skeletons [Col89], has studied how well a divide-and-conquer skeleton can be implemented on a grid architecture. He proposes to use an H-tree layout to map tree processors to mesh processors, but has not implemented the system. In terms of performance, he has focussed on the asymptotic efficiency with which a large grid of processors can implement the system with respect to the performance of a single processor. In contrast, the *TrEK* design can work on multicomputers with any topology and has been implemented on a transputer-based multicomputer. Performance models

derived in this thesis are analytical models, unlike Cole's asymptotic models, and hence can be used for performance tuning.

3.5 Chapter Summary

In this chapter, we have described an integrated approach for addressing programming and performance modeling problems on multicomputers. We have discussed the characteristics of two task-oriented paradigms, processor farm and divide-and-conquer, that are considered in this thesis. The following chapters describe the design and implementation of runtime systems, development of performance models and their experimental validation on a 75-node transputer based multicomputer. An integrated programming environment that includes programming tools such as graphical interface, mapper, loader, monitor and debugger in addition to virtual machines would make programming these machines much easier. Such an environment is described in Chapter 8.

Chapter 4

Processor Farm: Design and Modeling

In this chapter, we describe a processor farm and detail the trade-offs involved in its design. We derive models that accurately describe the behavior and performance characteristics of the system. We give the limitations and assumptions on which the models are based and describe how the models were used in the design process. The models are sufficiently general that they can be used to predict performance of our design on any topology. Providing independence from both size and topology while maintaining the ability to tune the performance strongly supports reuse.

In Section 4.1, we classify different types of processor farms and present our processor farm system, *Pfarm*. We compare and contrast our design with that of others at appropriate places within this section. In Section 4.2, we derive general analytical models that describe the start-up, steady-state, and wind-down phases of the execution on any tree topology. As balanced tree topologies provide maximum performance among all topologies of the same size, we apply this modeling technique to derive expressions for balanced tree topologies. We close by discussing how our models can be used in performance tuning and restructuring of application programs.

4.1 *Pfarm*: Design and Implementation

We begin by listing some of the important design issues and goals that must be addressed while designing and developing a system to execute applications efficiently using the processor farm strategy.

1. The system should fully exploit all the available parallelism in the hardware, such as the ability to simultaneously communicate on all links.
2. System overheads should be minimized.
3. The system should be topology independent, that is, it should scale and run on any processor topology.
4. The system should support reuse and provide an easy-to-use interface to the application programmer.

In a processor farm, control may either be centralized or distributed. In the case of centralized control, requests for work are sent to a central manager processor that assigns the tasks. Usually, as in Hey [Hey90], these requests are routed through the network back to the manager. However, it is also possible to dynamically reconfigure the links, with the use of crossbar switches, so that the manager can load and drain tasks directly from each worker [Hom93]. In the case of distributed control, there is a manager on each processor. In this case, the tasks flow into the system and the local manager either schedules an incoming task to the local worker process or forwards it to a child processor. Distributed control processor farms are common and have been described by many authors (for example see Cok [Cok91]).

Processor farms may be control driven or demand driven. A control driven scheme is useful when the work can be statically partitioned and assigned to the workers. A demand driven scheme, however, has the advantage that it can dynamically adjust to different sized tasks. Also, in a distributed processor farm, only neighbor to neighbor communication is necessary.

Pfarm implements a distributed demand-driven processor farm.

4.1.1 Process Structure and Scheduling

When implementing a distributed demand-driven processor farm, each worker processor consists of at least two processes: a task manager process and a worker process. Since intermediate processors in the network distribute tasks and collect results in addition to processing, from the performance point of view, it is important to overlap communication with computation. It affects the rate at which tasks can be forwarded, which, as shown in Section 4.2, limits the performance of the system. Although the processor farm implementations that have appeared in the literature [Cok91] mention the importance of overlapping communication with computation, most do not fully implement it.

In order to overlap communication with computation, it is necessary to have multiple processes on each worker processor. As a result, *Pfarm* has one InLink and one OutLink process for each hardware link in the processor, in case of transputers, there will in total be 4 InLink and 4 OutLink processes. The process structure of a worker with three children is shown in Figure 4.1. This figure depicts a single worker in the system. The entire system consists of a collection of these workers, organized in a tree structure, which logically corresponds to the ideal processor farm structure given in Figure 3.1. The result manager process in Figure 4.1 coordinates the collection and forwarding of results to the manager processor (or root) of the system. *Pfarm* takes advantage of the transputer's ability to use the links and the CPU simultaneously, and makes it possible to overlap computation with the transfer of tasks and results. Note that there is still a non-overlapped message start-up time associated with each communication.

Another important design consideration is the scheduling of local processes. In order to keep the worker processors busy processing tasks, it is important to forward the tasks as quickly as possible. Therefore, all the processes that distribute tasks should be run at high priority. *Pfarm* uses the hardware scheduler provided on the transputer chip. In *Pfarm*, all the task communicating link processes and the task manager process are run at high priority, whereas the worker process is run at low priority. In addition, link processes that forward results and the result manager are also run at high priority. As described in Section 4.2.3, this is important whenever system throughput is bound by the rate at which the manager receives the results. Also, this returns the results to the

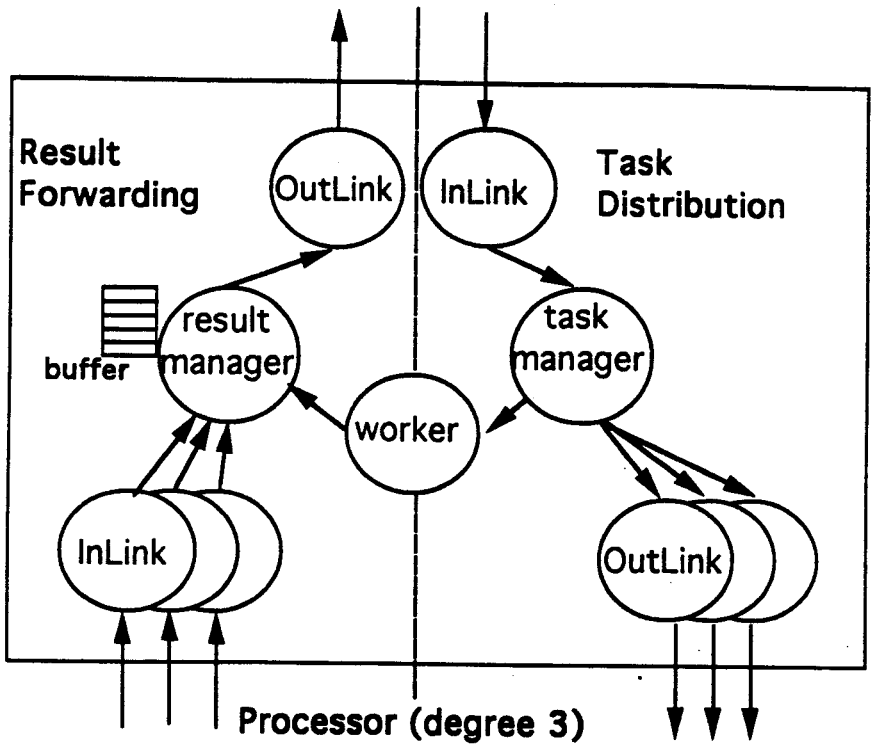


Figure 4.1: Process Structure on a Worker Processor in *Pfarm*

manager as quickly as possible which is important when there are dependencies among the tasks.

4.1.2 Task Scheduling

Since *Pfarm* distributes the control, there is a task manager process on each worker processor. The local manager gives priority to the local worker process while allocating an incoming task. If the local worker process is busy, the manager attempts to forward the task to one of its children using a round robin strategy among the free OutLink processes. The order in which the tasks are assigned to the OutLink processes depends on the underlying topology and affects the start-up time of the system. An analysis of the affect of round robin scheduling of tasks to OutLink processes during start-up is

given in Section 4.2.

In *Pfarm*, we initially flood-fill the system with tasks. However, once full, the system is demand-driven with new tasks entering the system as tasks are completed. This scheme works well for start-up and steady-state but is not as effective for the wind-down phase. During wind-down, there are no longer any incoming tasks and workers closer to the root may idle since remaining tasks are still being forwarded towards the workers farthest from the root.

4.1.3 Buffering

The number of task buffers to be allocated to each processor is an important design issue. In *Pfarm*, each process in the task distribution path shown in Figure 4.1 can have only one task. This provides sufficient buffering so that a process never idles waiting for a task. For example, when the worker process finishes, there will be another task available at the manager process. If the manager process was not there and the worker received the task directly from the InLink, then it would have to wait whenever the InLink process was in the midst of a communication. We call the task buffer in the manager process, an additional buffer since it is there for synchronization purpose only. It is important to minimize the number of buffers because the wind-down time increases proportionally with the number of buffers. This occurs because more tasks end up on the workers at the leaves of the tree, resulting in workers closer to the root idling (a complete analysis of wind-down is given in Section 4.2.1). The number of buffers adversely affects start-up as well, as we show in Section 4.2.1. However, the number of buffers does not affect steady-state performance.

At any given time, each processor can be viewed to have four tasks assigned to it, one each to the task manager process, local worker process, local InLink process and OutLink process of the *parent* processor. Thus, in total there are $4N$ active tasks in any N processor system, irrespective of the topology.

We do not restrict the number of result buffers as this does not affect the overall performance. But, at any given time, the number of active result buffers on any processor is small as the result forwarding communication processes and the result manager are

run at high priority.

In contrast, the amount of buffering required in a centralized scheme depends on the task size and message latency for receiving a new task. Unlike *Pfarm*, as N increases, message latency also increases and therefore the number of buffers per processor has to increase. In the centralized scheme, there is also the extra overhead of sending and receiving task request messages.

4.1.4 Topology Independence

Pfarm system is designed to be topology independent. For processors and links that satisfy the assumptions described in Section 3.2, the observations about the *Pfarm* design remain true, *independent* of the topology. Besides transputers, there are other machines such as iPSC hypercubes [Arl88] and TI C40 [Inc91] with similar behavior. Moreover, as we show in the next section, accurate performance models can be derived for *Pfarm* on any tree topology. For a given fixed interconnection, by taking a spanning tree, it is possible to use *Pfarm* and derive a model to predict its performance.

4.2 Performance Modeling

In summary, as a consequence of the design, *Pfarm* has the following characteristics:

1. The hardware system is a distributed memory message passing architecture with linear message cost model.
2. There is a continuous flow of tasks into the farm.
3. All tasks originate from a single source and results are returned to the source.
4. Tasks are dynamically distributed to the workers.

Our objective is to find a distribution of the load to all the worker processors so as to minimize the total execution time, where load consists of both the computational requirements of the tasks and the associated overheads for forwarding and executing the tasks.

The system can be either computation bound or communication bound. When it is computation bound, the system acts as a pipeline, with three phases to be analyzed: start-up, steady-state and wind-down. The start-up phase begins when the first task enters the system and ends when all the worker processors have received at least one task. After start-up, the system is in steady-state where it is assumed that the processors do not idle. Finally, the wind-down phase begins when the last task enters the system and ends when all the results have reached the source. The total execution time is given by,

$$T_{total} = T_{su} + T_{ss} + T_{wd} \tag{4.1}$$

where T_{su} is the start-up time, T_{ss} is the steady-state time and T_{wd} is the wind-down time. For a sufficiently large number of tasks, steady-state time dominates the remaining two phases. However, to better understand the limitations of processor farms with a smaller number of tasks, it is important to analyze start-up and wind-down. When the system is communication bound, the total execution time is determined by the rate at which either the tasks can be distributed to the worker processors or the results can be received from them.

In Sections 4.2.1 and 4.2.2, we derive performance models for the case in which the system is computation bound. In Section 4.2.1, we present a general analytical framework to analyze the steady-state performance of processor farms on any tree topology. We argue that, under reasonable assumptions, *Pfarm* obtains optimal performance on any topology. Later in this section, upper bounds for start-up and wind-down time are also derived. In Section 4.2.2, we derive steady state, start-up and wind-down models for balanced complete trees using the general analytical framework. Balanced complete trees are interesting because they provide maximum performance among all topologies of the same size. In Section 4.2.3, we analyze the performance of processor farms when they are communication bound.

4.2.1 General Analytical Framework

Let T be a tree architecture with processors $p_1, \ldots, p_N$. Let $C(i) = \{j \mid p_j \text{ is a child of } p_i\}$ denote the children of p_i in T. Let $\alpha = T_e + \beta_e$ be the processing

time plus associated overhead to execute a task locally, and let β_f be the processor overhead for every task forwarded to a child processor. β_f includes all the CPU overheads involved in receiving a task, forwarding it to a child processor, receiving the corresponding result and forwarding it to the parent processor. Let d and r be the average data and result size per task, respectively, and let τ be the communication rate of the links.

Steady-state Analysis

The steady-state phase begins once all the processors have a task to execute and ends when the last task enters the system. It is assumed that no processor idles during steady-state; a processor is either processing a task or busy forwarding tasks and results. Suppose that M tasks are executed during this phase and let V_i denote the number of tasks that visit p_i. Then, the following condition holds for all the processors in T,

$$T_{ss} = \alpha(V_i - \sum_{j \in C(i)} V_j) + \beta_f \sum_{j \in C(i)} V_j \tag{4.2}$$

$$= \alpha V_i + (\beta_f - \alpha) \sum_{j \in C(i)} V_j \tag{4.3}$$

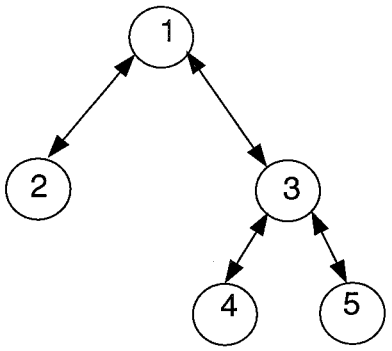
That is, the steady-state execution time (T_{ss}) equals the processing time with the associated overhead (α) for all the tasks executed locally plus the overhead (β_f) for all the tasks that were forwarded.

For a fixed T_{ss}, α, and β_f, these N conditions form a system of linear equations on N unknowns; $V_1, V_2, \ldots V_N$. By ordering the equations so that the parent of a processor in T appears before its children, it is easy to see that the system is in an upper triangular form. Thus, the N equations are linearly independent and there is a unique solution.

At the root of T, $V_1 = M$. Furthermore it is easily verified, by back substitution, that V_1 is a linear function of T_{ss}. Thus T_{ss} can be expressed in terms of M, α and β_f, which implies that given M, α and β_f, we can solve for T_{ss} and V_1 to V_N. We can also solve for f_i, the fraction of the M tasks executed by the ith processor,

$$f_i = \frac{1}{M}\left(V_i - \sum_{j \in C(i)} V_j\right).$$

In addition, we can obtain the steady-state throughput, the rate at which tasks can be processed by the system. An example of this analysis for an arbitrary architecture is given in Figure 4.2.



$$\begin{pmatrix} \alpha & \beta_f - \alpha & \beta_f - \alpha & & \\ & \alpha & & & \\ & & \alpha & \beta_f - \alpha & \beta_f - \alpha \\ & & & \alpha & \\ & & & & \alpha \end{pmatrix} \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \end{pmatrix} = \begin{pmatrix} T_{ss} \\ T_{ss} \\ T_{ss} \\ T_{ss} \\ T_{ss} \end{pmatrix}$$

$$T_{ss} = \frac{\alpha^3 M}{5\alpha^2 - 6\alpha\beta_f + 2\beta_f^2}$$

Figure 4.2: An example of the steady-state analysis

This analysis gives the execution time of the steady-state phase in terms of parameters that can be determined prior to execution. The total number of tasks, M, is usually known. α can be estimated or measured by using the techniques that are described in Chapter 8. The *Pfarm* system is designed such that the two overheads β_e and β_f are application and topology independent. Therefore, they need only be determined once for a particular implementation of *Pfarm*. In Section 5.1, we describe how the values of these overhead parameters can be determined.

As explained above, we can determine T_{ss} for a given arbitrary tree, T. However, for an arbitrary topology there remains the question of which spanning tree to use for *Pfarm*. We show that all shortest-path, demand-driven distribution schemes with the same overheads (β_e and β_f) are equivalent. Let $T_{ss}(S)$ be the execution time of a shortest-path, demand-driven distribution scheme S.

Theorem 1 *For any topology G, $T_{ss}(S)$ equals $T_{ss}(Pfarm(T))$ where $Pfarm(T)$ is Pfarm executing on T, a breadth-first spanning tree of G rooted at the source of the tasks.*

Proof:

Let $L(i)$ of G denote the set of processors that are at distance i from the source of the tasks. Since S and $Pfarm(T)$ are both shortest-path distribution schemes, tasks executed at a processor in $L(i)$ must have been forwarded from processors in $L(i-1)$. Let $s_i(S)$ and $s_i(T)$ denote the combined throughput of all the processors in $L(i)$ for scheme S and $Pfarm(T)$, respectively. We claim that for all i, $s_i(S) = s_i(T)$. It is initially true for n, the last level, since in both schemes the processors in $L(n)$ do not idle and can only execute tasks. In general, by induction on the level, the fact that processors do not idle and $s_i(S) = s_i(T)$ implies that both schemes must execute the same number of tasks on processors in $L(i-1)$. Thus $s_{i-1}(S) = s_{i-1}(T)$ and, in particular, $s_0(S) = s_0(T)$.

Therefore for a fixed M, since the throughput for both the schemes are the same, $T_{ss}(S) = T_{ss}(Pfarm(T))$.

□

It follows from Theorem 1 that for a fixed M, there is only one value of T_{ss} that ensures that processors do not idle. This does not exclude the possibility that there exists some non-shortest path scheme or a scheme that introduces idle time that would perform better. However, since this either increases the overhead to forward a task to a worker or reduces the computational power of a processor, it would be surprising if it outperformed the work efficient scheme we have analyzed.

Start-up and Wind-down Analysis

Start-up and Wind-down costs depend on the task distribution strategy and the hardware topology. For analyzing start-up and wind-down costs, we consider the structure of the underlying process graph. Given a tree architecture, the *process graph* of the system can be constructed by replacing each processor by the process structure given in Fig-

ure 4.1. We remove the processes that gather the results and only consider the processes that are involved in task distribution, that is, the worker process, the manager process and the link processes. The process graph of the architecture given in Figure 4.3(a) is shown in Figure 4.3(b). Notice that there are two types of edges in Figure 4.3(b): edges that represent the inter-processor communication and the intra-processor communication.

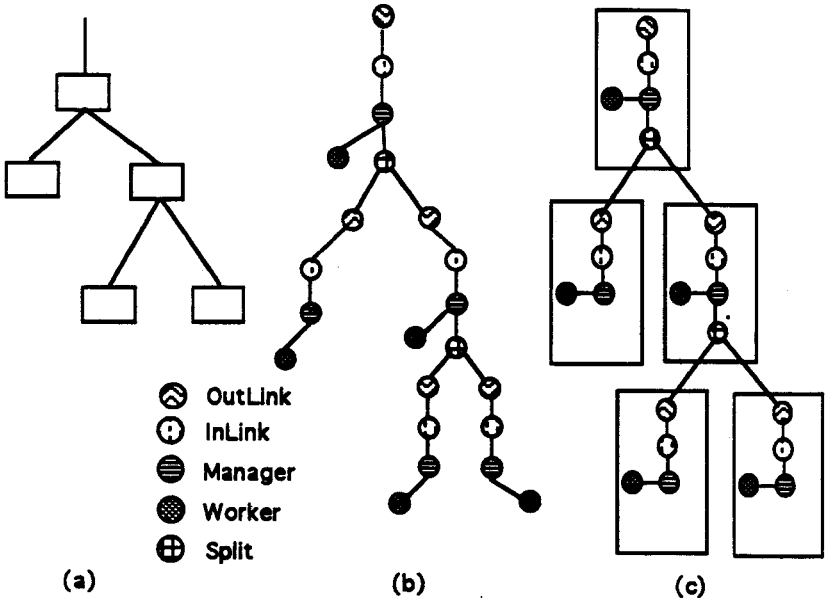


Figure 4.3: (a) node graph (b) process graph (c) subtree decomposition

Let T be the process graph of an N node architecture. We will add, as part of T, an initial OutLink process which we take as the root of T. In total, T has $4N$ processes or alternatively T can be viewed as consisting of N subtrees (or nodes) of the type depicted in Figure 4.3(c).

Start-up

Start-up begins when the first task enters the system and ends when all the processors

have received at least one task. The duration of the start-up phase depends on how the tasks are distributed to the processors. As mentioned in Section 4.1, the manager process gives priority to the worker process over the OutLink processes while allocating the tasks. Thus, when a worker process becomes free, it will receive the next available task. The OutLink processes, when free, receive tasks from the manager process; if more than one OutLink process is free, the manager allocates the tasks in a round robin ordering of the OutLink processes. Thus, as the tasks start entering the system, a processor keeps the first task it receives and then, as long as the worker process has not completed the current task, distributes the incoming tasks to its children.

In order to obtain an upper bound on start-up time, we discretize the start-up into a sequence of steps. On each step, a task is transferred from an OutLink process of one processor to either the worker process, or an OutLink process, or, when they all have tasks, to the last process along the path without a task. Furthermore, it is assumed that during the start-up phase, no worker process finishes its first task. This is a reasonable assumption since the task forwarding link processes run at high priority while the worker process runs at low priority. Assuming that there is a continuous flow of tasks into the system, we seek to bound the number of steps required before every worker process has a task.

For analysis purposes, we use a collapsed process graph like the one shown in Figure 4.3(c). This graph is identical to the original architecture graph except that each node consists of the InLink, the manager, and the worker process of a processor plus the corresponding OutLink process of its parent processor. In this tree, all the tasks are initially at the root and on each step every node with more than one task forwards, in round robin order, the last task it received to a child. In order to analyze the worst case, we assume that the tasks can be buffered at the manager process and thus each subtree has an infinite capacity to absorb tasks. We first obtain an upper bound on the number of steps to distribute at least one task to every node. This is equivalent to the procedure described above, except that it takes one additional step at the end to ensure that a leaf node forwards its task to the worker process.

Given a rooted oriented tree T, with child nodes numbered from left to right starting

at one, let $c(v)$ be the child number of node v with respect to the parent of v, $p(v)$. Let $p^n(v)$ denote the nth ancestor of node v in T, and let $\deg(v)$ be the down-degree of node v.

Definition 1 *Let* $d_n(v)$ *equal* $\prod_{i=0}^{n} \deg(p^i(v))$.

Lemma 1 *For any rooted oriented tree* T*, the first task received by node* v *is the*

$$1 + c(p^{n-1}(v)) + \sum_{j=0}^{n-2} c(p^j(v)) d_{n-j-2}(p^{j+2}(v))$$

task at the node $p^n(v)$*, the root of* T.

Proof:

Let $s(v,i)$ be the number of the ith task received by node v. Using the fact that the child $c(v)$ receives every $\deg(p(v))$ task arriving at $p(v)$, except for the first which is kept by $p(v)$, we obtain the following recurrence

$$s(v,i) = s(p(v), 1 + c(v) + (i-1)\deg(p(v))).$$

In general,

$$\begin{aligned}
s(v,i) &= s(p^1(v), 1 + c(p^0(v)) + (i-1)d_0(p(v))) \\
&= s(p(p(v)), 1 + c(p(v)) + [1 + c(v) + (i-1)d_0(p(v)) - 1]d_0(p(p(v))) \\
&= s(p^2(v), 1 + c(p^1(v)) + c(v)d_0(p^2(v)) + (i-1)d_0(p^1(v))d_0(p^2(v))) \\
&= s(p^2(v), 1 + c(p^1(v)) + \sum_{j=0}^{0} c(p^j(v)) d_{2-j-2}(p^{j+2}(v)) + (i-1)d_1(p(v))) \\
&\vdots \\
s(v,i) &= s(p^n(v)), 1 + c(p^{n-1}(v)) + \sum_{j=0}^{n-2} c(p^j(v)) d_{n-j-2}(p^{j+2}(v)) + (i-1)d_{n-1}(p(v)))
\end{aligned}$$

At the root, $s(v,i) = i$. Thus, when $p^n(v)$ is the root,

$$s(v,1) = 1 + c(p^{n-1}(v)) + \sum_{j=0}^{n-2} c(p^j(v)) d_{n-j-2}(p^{j+2}(v)). \tag{4.4}$$

□

Example:

Let us derive the task number of the first task that arrives at node 5 in the graph shown in Figure 4.3(a) using equation (4.4). For node 5, $n = 2$. Thus,

$$\begin{aligned} s(5,1) &= 1 + c(p^1(v)) + c(p^0(v))d_0(p^2(v)) \\ &= 1 + 2 + 2 \times 2 \\ &= 7. \end{aligned}$$

□

The time step at which the worker process on node v receives task $s(v,1)$ is $n + s(v,1)$. This follows from the fact that task $s(v,1)$ leaves the root after $s(v,1) - 1$ steps, and it takes $n + 1$ more steps for this task to reach the worker process on node v as this task gets forwarded in every step because $s(p^i(v),1) < s(v,1)$. The next theorem follows from these remarks.

Theorem 2 *For any tree structured process graph, after*

$$\max_{v \text{ a leaf}} \{n + s(v,1)\}$$

steps, every worker process has a task to execute.

The time required for each step is determined by the communication cost to transfer a task from a processor to its child and the associated CPU overhead. The average communication time needed to transfer a task from one processor to its child is given by $T_{cd} = d/\tau$. The associated overhead is $\beta_f/2$ since β_f includes the overheads for both transferring a task to a child node and returning the corresponding result to the parent. Therefore, the time required for each step is given by $T_{cd} + \beta_f/2$.

From Theorem 2 it follows that

$$T_{su} = (T_{cd} + \beta_f/2) \max_{v \text{ a leaf}} \{n + s(v,1)\} \tag{4.5}$$

For the tree shown in Figure 4.3(a), the start-up time is determined by the time required for node 5 to receive its first task. As derived earlier, $s(5,1) = 7$ and $n = 2$.

Thus, the start-up time for this example is given by

$$T_{su} = 9(T_{cd} + \beta_f/2) \tag{4.6}$$

The upper bound on the number of steps required for every worker process to have a task to execute can be exponential in N. Consider for example a tree with a long path where each node on the path has large degree but all nodes off of the path are leaves. The sum of products in $s(v, 1)$ grows exponentially with respect to N. This is a result of our assumption that subtrees can always accept tasks. In practice, the upper bound cannot exceed the number of buffers in the tree, $4N$. But it is possible to come arbitrarily close to $4N$. As the upper bound is proportional to the number of buffers, for start-up, ideally the number of buffers should be minimized. This is one of the reasons we chose to have only one additional buffer on every worker processor.

Although the degree and depth of the tree is fixed, it is possible to change $c(v)$. Theorem 2 provides a means for determining an orientation of the tree that minimizes start-up time.

Example:

Earlier, for the tree shown in Figure 4.3(a), we found that $s(5, 1) = 7$. If we change the orientation of the tree by interchanging nodes 2 and 3, the start-up time is still determined by the time required for node 5 to receive its first task. Now, however $s(5, 1) = 6$.

□

In summary, to minimize start-up, the tree should be oriented so that the longest path appears on the left (that is, on start-up, tasks are forwarded first along the longer paths).

Wind-down

Wind-down begins when the last task enters the system and ends when the last result leaves the system. The Wind-down phase can be broken into two parts: the time to complete the remaining tasks in the system and the time to return results that are in the system after all the tasks have been executed.

Consider the state of the process graph when the last task enters the system. As given in Section 4.1, there are $4N$ tasks. In order to derive an upper bound, let us

assume that all the remaining tasks have just started to execute. We will bound the maximum number of tasks executed by a single processor during the wind-down phase. This gives an upper bound on the time taken to complete the $4N$ tasks.

If all of the tasks have just begun execution, then after time α, each processor has executed one task. Since priority is given to distributing the tasks, some of the tasks buffered in each processor will be forwarded to the child processors. In the worst case, when α greatly exceeds β_f, the tasks will be forwarded as far as possible towards the leaves.

A worker process at a leaf can only execute those tasks available at its ancestor processes in the process graph. Therefore, we should consider the longest path from the root to a leaf in the tree (this is at most $3N + 1$, for example see Figure 4.3(c)) to derive an upper bound. Let m be the number of ancestor processes of the leaf process in the longest path, each of which initially contains a task. Starting at the root, every third process along the path is adjacent to a worker process. Therefore, after time α, when each of the worker processes have finished executing a task, each worker process adjacent to the path will receive a task from a manager along the path. The remaining tasks on the path shift down towards the leaves filling as many buffers as possible. This results in the following recurrence for $p(n)$, the length of the path,

$$\begin{aligned} p(0) &= m \\ p(n) &= p(n-1) - \left\lceil \frac{p(n-1)}{3} \right\rceil . \end{aligned}$$

Solving this recurrence for $p(n) = 1$ shows that after $\lceil \log_{3/2} m \rceil + 1$ steps, all tasks have been processed. Thus, the time taken for the first part of the wind-down phase is given by $\alpha(\lceil \log_{3/2} m \rceil + 1)$.

The second part of the wind-down cost is determined by the time required to forward the last result from the leaf process to the root. If r is the average result size per task, the communication time needed to transfer a result from one processor to its parent is given by $T_{cr} = r/\tau$. The associated overhead per transfer is $\beta_f/2$ since β_f includes the overheads for both transferring a task to a child node and returning the corresponding result to the parent node. Therefore, the second part of the wind-down cost is given by

$m/3 \times (T_{cr} + \beta_f/2)$ since there are $m/3$ processors in the path. If m is the length of the longest path, the wind-down cost is given by

$$T_{wd} = \alpha(\lceil \log_{3/2} m \rceil + 1) + \frac{m}{3}(T_{cr} + \beta_f/2). \tag{4.7}$$

In the tree shown in Figure 4.3(a)), there are two longest paths, the path from the root to node 4 and the path from the root to node 5. In this example $m = 9$ and the wind-down time is given by

$$T_{wd} = 7\alpha + 3(T_{cr} + \beta_f/2). \tag{4.8}$$

This analysis depends only on the depth of the tree and therefore gives the same bound for all breadth-first spanning trees of the topology. Note, however, that the analysis is overly pessimistic since subtrees along a path from the root to a leaf also receive tasks from managers on the path. The actual wind-down time also depends on the number of nodes along the path with down degree greater than one. The fewer the number of nodes of degree one, the smaller is T_{wd}. Because of our round robin scheduling policy, any node of degree greater than one can only forward at most one task towards the leaf of the chosen path out of every three tasks that arrive at the node. Therefore, the number of tasks on the path decreases more quickly and if every processor on the longest path has at least two children, then the number of tasks executed by a leaf is bounded by $\max\{\lceil \log_3 m \rceil + 1, 4\}$. Leaves must execute at least 4 tasks, namely those in their local buffers.

For the example topology shown in Figure 4.2, the total execution time for processing M tasks is given by

$$T_{total} = 9(T_{cd} + \beta_f/2) + \frac{\alpha^3(M-20)}{5\alpha^2 - 6\alpha\beta_f + 2\beta_f^2} + 7\alpha + 3(T_{cr} + \beta_f/2). \tag{4.9}$$

4.2.2 Balanced Tree Topologies

In this section, we analyze the performance of processor farms on balanced tree topologies using the framework given in the previous section. Balanced tree topologies are of interest because (a) a k-ary balanced tree topology, where k is the number of links on each node,

provides optimal performance, and (b) for balanced trees, it is possible to obtain closed form solutions for system throughput and speedup.

For processor farms, a k-ary balanced tree topology provides optimal performance among all the topologies of the same size for the following reasons:

1. From Theorem 1, we know that the best topology for processor farms is a breadth first spanning tree. As a k-ary balanced tree is a spanning tree with minimum possible depth among all degree k graphs, it provides a steady state performance that is as good as any possible spanning tree.

2. As explained in Section 4.2.1, start-up cost is proportional to the length of the longest path in the graph. It also depends on the ordering of the children, and can increase when the graph is unbalanced. The length of the longest path in a k-ary balanced tree topology is minimum among all the graphs with the same number of nodes. The symmetry in balanced tree implies that the orientation of the tree does not affect the start-up cost. Thus, a balanced k-ary tree topology has minimum possible start-up time among all trees with the same number of nodes.

3. As shown in Section 4.2.1, wind-down cost is proportional to the length of the longest path in the graph. The wind-down cost also decreases as the number of children at processors in the longest path increases since these children steal tasks from the path. As a result, this decreases the number of tasks forwarded to the leaf processor in the longest path. As mentioned earlier, the length of the longest path in a k-ary balanced tree is minimum among all trees of the same size. In addition, all the non-leaf nodes in a balanced k-ary tree have the maximum possible number of children and thus reduce the number of tasks forwarded to any particular leaf. Thus, a complete k-ary tree has minimum possible wind-down time among all the trees with the same number of nodes.

As defined in Section 4.2.1, let α be the average processing time plus the overhead to execute a task locally and let β_f be the overhead for every task forwarded to a child processor. Consider a D level balanced k-ary tree with k^i processors on level i. Figure 4.4 shows binary and ternary tree topologies with $D = 4$ and 3 respectively.

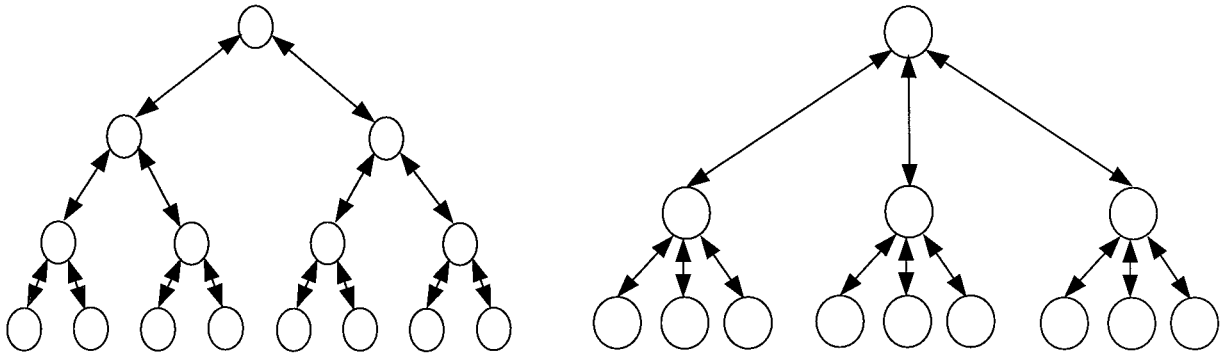


Figure 4.4: Binary and Ternary Trees with $D = 4$ and 3

We begin by analyzing the steady-state phase, followed by an analysis of start-up and wind-down.

Steady-state Analysis

Let M be the total number of tasks processed during the steady-state phase. Assuming that no processor idles during the steady-state, all the processors at a particular level of the tree execute the same number of tasks. We can then express the steady-state time (T_{ss}) for each processor in terms of the number of tasks processed and forwarded and their associated costs and overheads. From the general analysis in Section 4.2.1, we have

$$T_{ss} = \alpha(V_i - \sum_{j \in C(i)} V_j) + \beta_f \sum_{j \in C(i)} V_j \tag{4.10}$$

Since all the processors on a level execute same number of tasks, by summing equation (4.10) over all the processors on level i, we have

$$\begin{aligned} k^i T_{ss} &= \alpha(\sum_i V_i - \sum_i \sum_{j \in C(i)} V_j) + \beta_f \sum_i \sum_{j \in C(i)} V_j \\ &= \alpha \sum_i V_i + (\beta_f - \alpha) \sum_{i+1} V_j. \end{aligned} \tag{4.11}$$

Let

$$L_i = \frac{1}{k^i} \sum_i V_i$$

be the number of tasks that visit a processor on level i.
Therefore,

$$T_{ss} = \alpha L_i + (\beta_f - \alpha) k L_{i+1}.$$

By rearranging the above, we have

$$k(\alpha - \beta_f) L_{i+1} = \alpha L_i - T_{ss} \tag{4.12}$$

and $L_0 = M$.

The above recurrence is of the following form described in Chapter 2 of Knuth's Concrete Mathematics [GKP89].

$$\begin{aligned} a_n &= k(\alpha - \beta_f) \\ b_n &= \alpha \\ c_n &= -T_{ss} \\ s_n &= \left[\frac{k(\alpha - \beta_f)}{\alpha}\right]^{n-1}. \end{aligned}$$

Solving the recurrence, we obtain

$$L_i = M\left[\frac{\alpha}{k(\alpha - \beta_f)}\right]^i - T_{ss}\left(\frac{\alpha^{i-1}}{[k(\alpha - \beta_f)]^i}\left[\frac{1 - \left[\frac{k(\alpha-\beta_f)}{\alpha}\right]^i}{1 - \frac{k(\alpha-\beta_f)}{\alpha}}\right]\right).$$

Let

$$a = \frac{\alpha}{k(\alpha - \beta_f)}.$$

Then

$$\begin{aligned} L_i &= Ma^i - \frac{T_{ss}}{\alpha} a^i \left(\frac{1 - 1/a^i}{1 - 1/a}\right) \\ &= Ma^i - \frac{T_{ss}(a^i - 1)}{\alpha(1 - 1/a)}. \end{aligned} \tag{4.13}$$

Since $L_i = 0$ for $i = D$,

$$\begin{aligned} Ma^D &= \frac{T_{ss}(a^D - 1)}{\alpha(1 - 1/a)} \\ T_{ss} &= \frac{M\alpha(1 - 1/a)}{(1 - 1/a^D)}. \end{aligned}$$

By substituting for a, we obtain

$$T_{ss} = \frac{M[\alpha - k(\alpha - \beta_f)]}{1 - \left(\frac{k(\alpha-\beta_f)}{\alpha}\right)^D}. \tag{4.14}$$

Discussion

From the steady-state execution time given by equation (4.14), we can derive expressions for throughput and speedup. The steady-state throughput of a D level balanced k-ary tree is given by

$$\begin{aligned} S_D &= \frac{M}{T_{ss}} \\ &= \frac{1}{\alpha - k(\alpha - \beta_f)} \left[1 - \left(\frac{k(\alpha - \beta_f)}{\alpha}\right)^D\right]. \end{aligned} \tag{4.15}$$

The steady-state speedup of a D level balanced k-ary tree is given by

$$\begin{aligned} SP_D &= \frac{M\alpha}{T_{ss}} \\ &= \frac{\alpha}{\alpha - k(\alpha - \beta_f)} \left[1 - \left[\frac{k(\alpha - \beta_f)}{\alpha}\right]^D\right]. \end{aligned} \tag{4.16}$$

Here, speedup* is defined as the ratio of the execution time of the parallel algorithm on a single processor (execution time includes the associated overhead in addition to task processing time) to the total execution time of the algorithm on the parallel system.

We can also determine the fraction of the total number of tasks that are executed on each processor. For a processor on level i, it is given by

$$f_i = \frac{L_i - kL_{i+1}}{M}. \tag{4.17}$$

By substituting equation (4.14) for T_{ss} in equation (4.13),

$$\begin{aligned} L_i &= Ma^i - M(a^i - 1)\left[\frac{a^D}{a^D - 1}\right], \\ \frac{L_i}{M} &= a^i - (a^i - 1)\left[\frac{a^D}{a^D - 1}\right], \\ &= \frac{a^D - a^i}{a^D - 1}. \end{aligned}$$

*This is different from the usual theoretical measure of speedup, the time of the most effective serial algorithm divided by the time of the parallel algorithm.

By substituting the above in equation (4.17),

$$\begin{aligned} f_i &= \frac{a^D - a^i}{a^D - 1} - k\frac{a^D - a^{i+1}}{a^D - 1}, \\ &= \frac{a^D(1-k) + a^{i+1} - a^i}{a^D - 1}. \end{aligned} \tag{4.18}$$

Now consider the affect of β_f on the overall performance. Let g be equal to the ratio β_f/α. This is the inverse of granularity which is generally defined as the ratio of computation to communication overhead. The speedup given in equation (4.16) can now be expressed in terms of g as follows:

$$\begin{aligned} SP_D &= \frac{\alpha}{\alpha - k(\alpha - \beta_f)}\left[1 - \left(\frac{k(\alpha - \beta_f)}{\alpha}\right)^D\right] \\ &= \frac{1 - [k(1-g)]^D}{1 - k(1-g)} \\ &= \sum_{i=1}^{D} [k(1-g)]^{i-1}. \end{aligned}$$

When $g \to 0$, we have

$$SP_D = \frac{1 - k^D}{1 - k} = N,$$

the total number of processors (i.e., when overhead β_f is negligible compared to α, speedup is proportional to the number of processors in the system).

When $g \to 1$ (i.e., when overhead β_f is almost equal to α), $SP_D = 1$. For the case $0 < 1 - g < 1$, $(1 - g)$ is the factor by which the processing capacity of a processor is reduced due to the overheads involved in dynamically distributing the work. Figure 4.5 shows how overhead β_f affects the efficiency.

In case of a linear chain of N nodes, speedup is given by

$$SP_N = \frac{1 - (1-g)^N}{1 - (1-g)} = \frac{1}{g} - \frac{(1-g)^N}{g} \tag{4.19}$$

For large N,

$$SP_N = \frac{1}{g}$$

as the second term in equation (4.19) becomes negligible. Thus $1/g$ gives an upper bound on speedup on a linear chain.

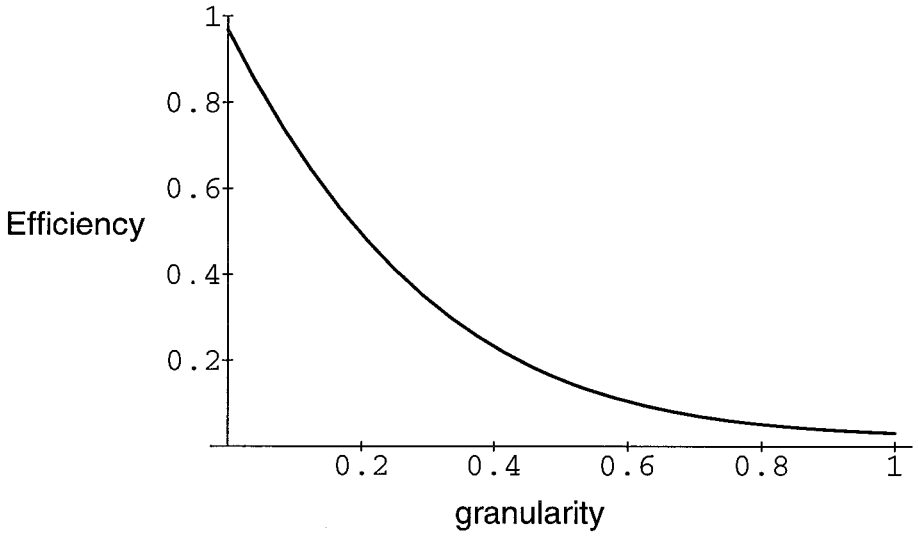


Figure 4.5: The affect of β_f on efficiency

The analytical results derived here for k-ary balanced trees are similar to those obtained in [Pri87, Pri90, TD90]. The difference is in the way the overhead parameters are included in the model. With proper substitutions, it is possible to obtain their throughput expressions from our model. Pritchard has analyzed processor farm on a linear chain and his model [Pri87, Pri90] is an abstract one which uses the characteristics of the machine as the overhead parameters to the model. As a result, it does not take into account the scheduling strategy and the associated software overheads. Tregidgo and Downton [TD90] have extended Pritchard's analysis for balanced binary and ternary trees. Again, these models only considered the hardware characteristics as the overhead parameters. Tregidgo and Downton have validated their model using a simulator. Contrary to statements in [TD90], the model they derived also holds for distributed farms as well as small centralized farms.

In comparison, we have provided a general framework to derive the performance models for processor farms on any topology. These models assume a realistic dynamic scheduling strategy, and account for all the associated software overheads. Also, we have analyzed start-up and wind-down phases, which are significant for applications consisting

of a smaller number of tasks. The models have been experimentally validated and are accurate as discussed in Chapter 5.

Start-up Analysis

The start-up analysis presented in Section 4.2.1 for arbitrary tree topologies can be used to obtain start-up costs for balanced tree topologies. The start-up cost for an arbitrary topology is given by equation (4.5), which is reproduced below,

$$T_{su} = (T_{cd} + \beta_f/2) \max_{v \text{ a leaf}} \{n + s(v,1)\}. \tag{4.20}$$

In a balanced tree, the rightmost leaf node will be the last among all the nodes to receive its first task. Thus, the start-up time is determined by the number of steps required for the rightmost leaf node to receive its first task. For a D level, k-ary tree, $n = D-1$. For the rightmost leaf node, $s(v,1)$ can be obtained using equation (4.4), and is given by

$$s(v,1) = 1 + k + k^2 + \ldots + k^{D-1} = \frac{k^D - 1}{k-1} = N,$$

the total number of nodes.

Thus, for a balanced k-ary tree of D levels, the start-up cost is given by

$$T_{su} = (N + D - 1)(T_{cd} + \beta_f/2). \tag{4.21}$$

Wind-down Analysis

In this section, we use the wind-down analysis presented in Section 4.2.1 for arbitrary tree topologies to derive the wind-down time for balanced trees. The wind-down time is given by equation (4.7) which is reproduced below:

$$T_{wd} = \alpha(\lceil \log_{3/2} m \rceil + 1) + \frac{m}{3}(T_{cr} + \beta_f/2). \tag{4.22}$$

where m is the length of the longest path in the process graph. For a N node linear chain topology, the wind-down analysis presented in Section 4.2.1 holds with $m = 3N$. The wind-down time is given by

$$T_{wd} = \alpha(\lceil \log_{3/2} 3N \rceil + 1) + N(T_{cr} + \beta_f/2). \tag{4.23}$$

As explained in Section 4.2.1, in a balanced tree with degree greater than one, the number of tasks in the longest path decreases more quickly. For this case, the number of tasks executed by the leaf node on the longest path is given by $\max\{\lceil \log_3 m \rceil + 1, 4\}$. For a D level k-ary balanced tree, $m = 3D$, and wind-down cost is given by

$$T_{wd} = \alpha(\lceil \log_3 3D \rceil + 1) + D(T_{cr} + \beta_f/2). \tag{4.24}$$

The total execution time for processing M tasks on a D level k-ary balanced tree is given by

$$\begin{aligned} T_{total} &= T_{su} + T_{ss} + T_{wd} \\ &= (N + D - 1)(T_{cd} + \beta_f/2) + \frac{(M - 4N)[\alpha - k(\alpha - \beta_f)]}{1 - \left(\frac{k(\alpha-\beta_f)}{\alpha}\right)^D} \\ &\quad +\alpha(\lceil \log_3 3D \rceil + 1) + D(T_{cr} + \beta_f/2). \end{aligned}$$

4.2.3 Communication Bound

The performance models derived in Sections 4.2.1 and 4.2.2 are applicable only when the system is computation bound. If the system is communication bound, it may never reach steady-state and processors may idle. In this section, we analyze the performance of processor farms, when the system is communication bound.

There are two cases in which the performance of a processor farm system might be communication bound. The first corresponds to the actual transfer portion of the communication; the second corresponds to the CPU overhead required for communication. Here, we assume that the links are homogeneous.

Case (i)

In the first case, throughput is limited either by the the rate at which the farm can receive tasks or the rate at which it can transfer results to the manager, whichever is smaller. Let

$$T_c = \max\{T_{cd}, T_{cr}\}.$$

The system throughput corresponding to this limit is

$$S_{com1} = \frac{1}{T_c + \beta_c}, \tag{4.25}$$

where β_c is the processor overhead required to receive a task from a parent or to send a result to the parent. β_c includes the overheads required to make the newly arrived task available for processing (or forwarding), to allocate a new buffer for the Inlink process to receive the next task and to initiate the communication. It also represents the corresponding time required to initiate the transfer of a new result after the communication of a previous result to the parent node is completed. In addition, overhead β_c can be estimated by $\beta_c = \beta_f/4$ as β_f is the total processor overhead for receiving a task, forwarding it to a child processor, receiving the corresponding result and forwarding the result to the parent node.

Case (ii)

The second factor that limits throughput is the CPU overhead in transferring the tasks and results. Since the first worker processor in the farm has to incur an overhead of at least β_f for every task received from the manager and forwarded to a child processor, the overall throughput is limited by

$$S_{com2} \leq \frac{1}{\beta_f}, \tag{4.26}$$

irrespective of the number of workers in the farm.

The communication bound on the throughput of the system is the smaller of the two bounds obtained from equations (4.25) and (4.26).

4.3 Discussion

In this section, we discuss how the performance models derived in Section 4.2 can be used in performance tuning.

4.3.1 Optimal N and Topology

Our models can be used to determine the optimal N and topology to maximize performance for a given application program.

If $\beta_f < (T_c + \beta_c)$, then the intersection of equations (4.15) and (4.25) gives the optimal number of processors to use to maximize throughput. Beyond this optimal

number, overall performance of the system does not increase. For this case, ignoring start-up and wind-down, the optimal level of a k-ary processor tree, D_{opt}, is given by

$$D_{opt} = \frac{\log\left[1 - \frac{\alpha - k(\alpha - \beta_f)}{T_c + \beta_c}\right]}{\log\left[\frac{k(\alpha - \beta_f)}{\alpha}\right]}. \tag{4.27}$$

If $\beta_f > (T_c + \beta_c)$, then the optimal number of processors is given by the intersection of equations (4.15) and (4.26). For this case, D_{opt} is given by

$$D_{opt} = \frac{\log\left[1 - \frac{\alpha - k(\alpha - \beta_f)}{\beta_f}\right]}{\log\left[\frac{k(\alpha - \beta_f)}{\alpha}\right]}. \tag{4.28}$$

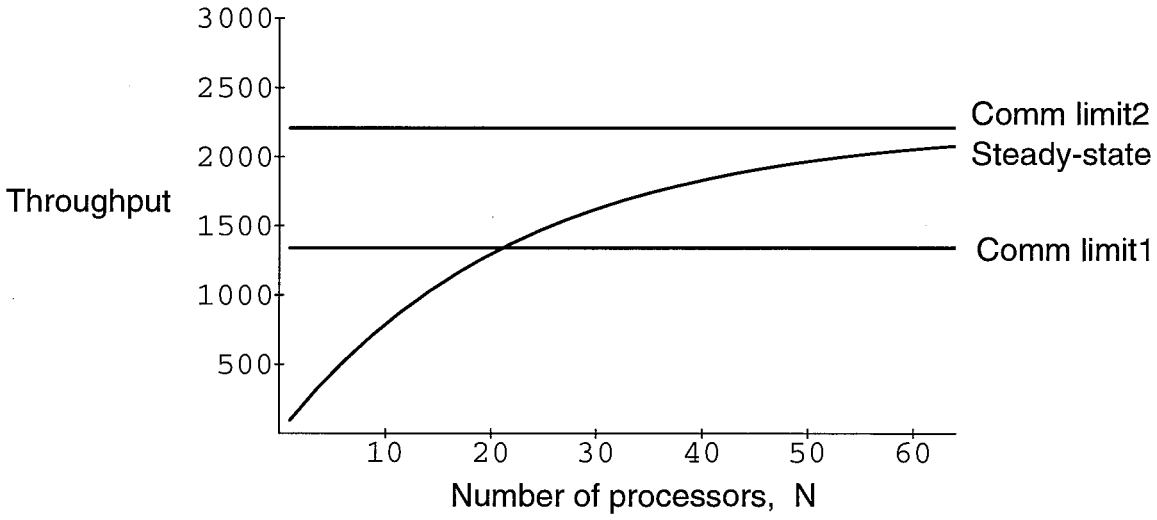


Figure 4.6: Plot of throughput curves for a linear chain (with $T_e = 10$ms, $\beta_e = 482\mu$s, $\beta_e = 453\mu$s)

For a linear chain of N nodes, steady-state throughput is given by

$$S_N = \frac{1}{\beta_f}\left[1 - \left(\frac{(\alpha - \beta_f)}{\alpha}\right)^N\right]. \tag{4.29}$$

In Figure 4.6, we have plotted the three throughput equations (4.25), (4.26) and (4.29) for a linear chain with a set of typical parameter values. As $\beta_f < (T_c + \beta_c)$ in this example, optimal N can be obtained from equation (4.27) which gives a value of 20.

Equation (4.28) is not applicable for linear chain even if $\beta_f > (T_c + \beta_c)$. For this case, As $N \rightarrow \infty$, throughput reaches the limit $1/\beta_f$. But from Figure 4.6, we can observe that after a certain value of N, the increase in throughput with an increase in the number of processors is very small. This value of N can be determined by iteratively evaluating the throughput for increasing N using equation (4.29).

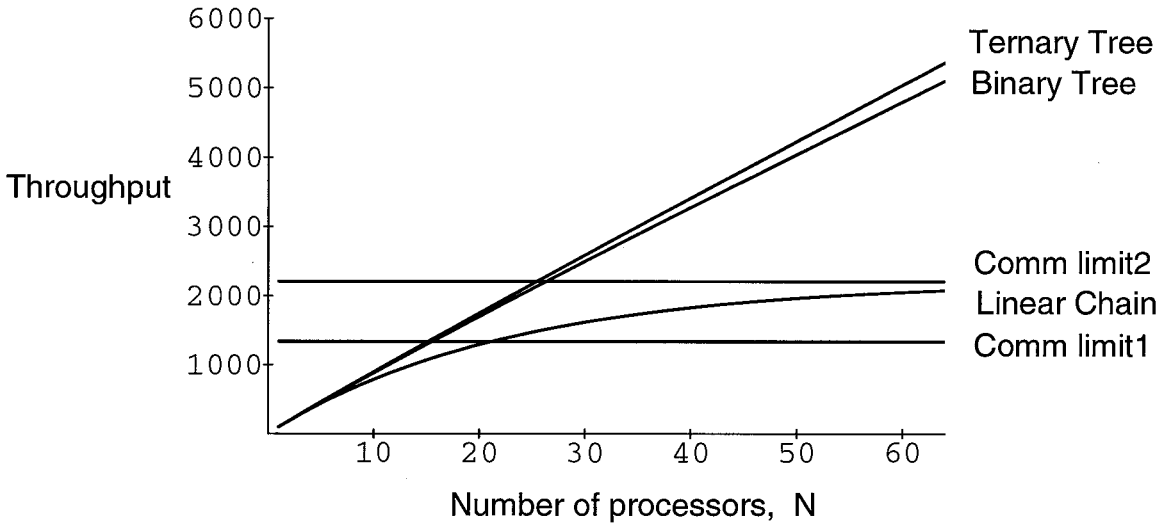


Figure 4.7: Comparison of processor farm throughput on linear chain, binary tree and ternary tree configurations

In Figure 4.7, we have plotted throughput as a function of the number of nodes for three different topologies: linear chain, binary and ternary tree. As expected, we can observe that for any particular N, the ternary tree configuration gives better throughput than the other two, as long as the system has not reached one of the two communication bounds. This shows that with a k-ary tree topology, it is possible to achieve the same throughput with fewer number of nodes. It also shows the dramatic increase in throughput possible by using a binary tree rather a chain. The increase in throughput from binary to ternary tree is much smaller.

If the total number of processors available in the system is less than the optimal number and it is not possible to have a complete k-ary tree, it is better to use a topology

in which all the levels except the last are complete k-ary with the remaining nodes balanced at the last level. If it is not possible to obtain a k-ary tree due to system configuration restrictions, it is better to use a tree with the next possible larger degree.

If the total number of processors available in the system is larger than the optimal number of nodes for a given application program, then one can use multiple k-ary tree topologies to improve performance. This is possible only if there are multiple links from the manager to the worker farm. The manager could be a host workstation or one of the multicomputer nodes. In the first case, the number of k-ary tree topologies one can use is limited by the number of available host links. In the second case, it is possible to have up to k k-ary tree topologies. For both cases, overall throughput of the system is given by the sum of the throughput of each of the individual tree topologies, provided the manager can keep up a continuous flow of tasks to all the worker trees.

In practice, throughput can decrease as N exceeds the optimal value. We have observed this phenomenon in the validation experiments and the reasons are discussed in Section 5.3.

4.3.2 Problem Scaling

Our models can be used to determine how well the speedup scales with problem size. In the processor farm case, an application can be scaled in two different ways. One way to scale the problem is to increase the total number of tasks, M. It follows from equation (4.16) that steady-state speedup is independent of M. Thus, scaling the problem size by increasing M does not lead to any increase in the steady-state speedup. However, it may lead to a small increase in the overall speedup because wind-down and start-up now represents a smaller portion of the total execution time.

The second way to scale a problem is to increase the granularity of the tasks by increasing T_e. In Figure 4.8, we have plotted steady-state speedup of a linear chain topology for different values of T_e. Figure 4.8 shows that steady-state speedup increases with increasing values of T_e as long as the system does not reach either of the two communication bounds. Thus, to increase speedup, it is better to increase T_e rather than M.

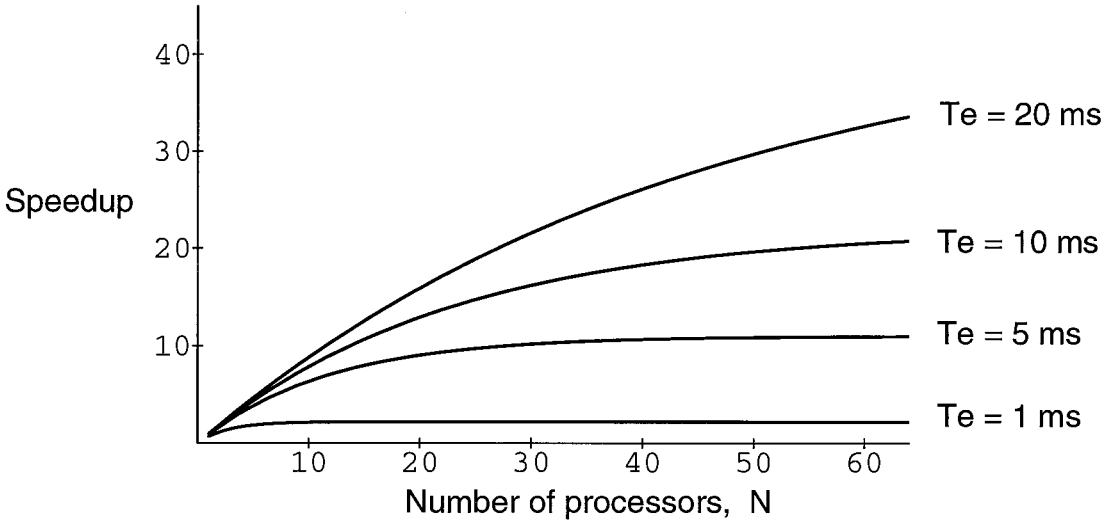


Figure 4.8: Measured speedup for processor farm on linear chain

4.3.3 Granularity

There are many applications in areas such as numerical analysis and image processing in which it is possible to decompose a problem of fixed size in several ways. The computation requirements of the tasks and the total number of tasks may vary from one case to another. Also, some programs may be easily restructured to produce tasks of different computation requirements. Granularity of the tasks is given by the computation requirements of the tasks. Performance models can be used to determine the best granularity to be used for an application to obtain maximum performance.

It can be observed from Figure 4.8 that for a fixed N, steady-state speedup increases with an increase in T_e. Also, the optimal value of N increases with increasing T_e. For a problem of fixed size, increasing the granularity reduces the total number of tasks, M. In addition, it may increase data and result sizes which leads to increased communication costs. Both small M and larger communication costs will add to start-up and wind-down costs.

Figure 4.9 shows a graph of speedup, with the effect of start-up and wind-down included, as a function of granularity and N for a processor farm running on a linear

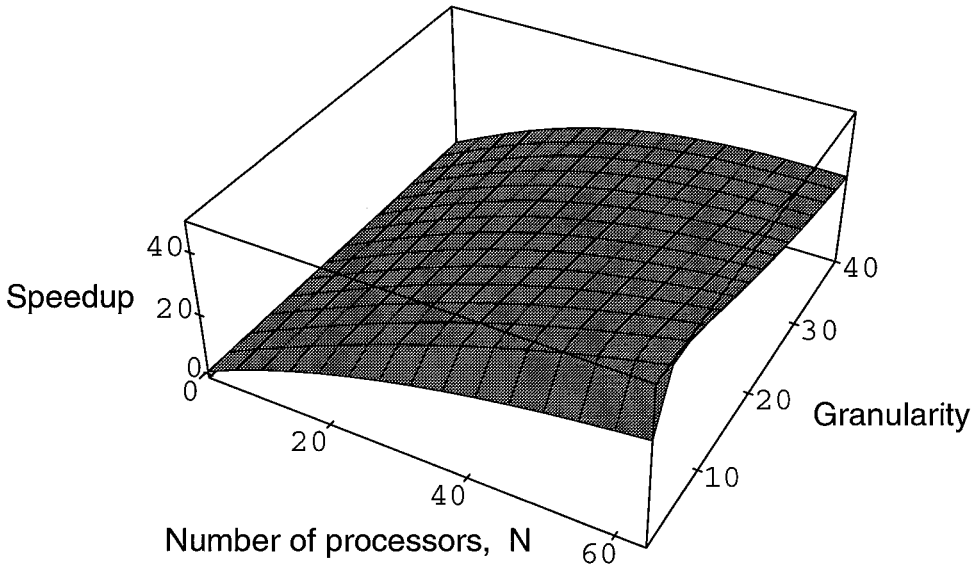


Figure 4.9: The affect of granularity on speedup

chain for a problem of fixed size. Here, granularity is defined as the ratio of new T_e to an original smaller T_e. Notice that, up to certain point, speedup increases with granularity, and then starts decreasing. Therefore, one can not use an arbitrarily large granularity, rather the optimal operating point of the system must be calculated as a function of N, T_e and the values of the overhead parameters.

4.4 Chapter Summary

In designing an efficient processor farm system, many trade-offs have to be considered. In this chapter, we have described the design of *Pfarm*, detailing the factors that affect the overall performance of the system and how they have to be addressed in the processor farm design. We have presented a general analytical framework that can be used to determine the performance of a processor farm system on any topology. The

interaction between the design and modeling phases have been discussed throughout the chapter. We have outlined how the models can be used in restructuring applications and in determining the optimal number of nodes and topology to be used to maximize performance.

In Chapter 5, we experimentally validate the performance models derived in this chapter on a large transputer-based multicomputer. The research results described in this chapter along with the experimental validation presented in the next chapter were published in [SCW92, WSC93].

Chapter 5

Processor Farm: Experiments

In this Chapter, we validate the performance models derived in Chapter 4 for processor farms. The models are experimentally validated using a *Pfarm* implementation on the multicomputer described in Section 3.3. *Pfarm* was validated using the Logical Systems version of the software.

We used a synthetic workload in all of the validation experiments. The application program consisted of a set of tasks, each of which executed an empty loop. The number of iterations of this loop determines the task execution time T_e for the particular experiment. By running the loop at high priority, it was possible to determine the number of iterations necessary to produce a task size of 1 ms. Multiples of this value were then used to obtain the different T_e's.

In Section 5.1, we describe the experiments conducted to determine the values of the system overhead parameters. In Section 5.2, we validate the performance models for arbitrary tree topologies, and in turn show that *Pfarm* works on an arbitrary topology. Performance models for balanced tree topologies are validated in Section 5.3. We describe the results of the experiments to test the robustness of our models in Section 5.4.

5.1 Determining System Overheads

In order to compare the analytical model with the actual execution, it is first necessary to determine the values of the system overhead parameters, β_e and β_f. As explained in

Section 4.2, β_e is the processor overhead to execute a task locally and β_f is the processor overhead for every task forwarded to a child processor. These overheads depend only on the implementation of *Pfarm* and are independent of the application program. Also, they do not depend on the underlying topology because the software paths in *Pfarm* that constitute these overheads are the same for any topology. Therefore, these overheads have to be determined only once for a particular *Pfarm* implementation. In some applications, it may be difficult to distinguish between what constitutes the computation time of a task (T_e) and the associated overhead (β_e). For these cases, we can measure α (the sum of T_e and β_e) and use it in the models. The techniques for measuring the values of α are discussed in Section 8.2.

The values of the overhead parameters, β_e and β_f, can be determined by running a few experiments on configurations with one and two worker processors (see Figure 5.1). These experiments were first conducted with the Logical Systems implementation of *Pfarm*.

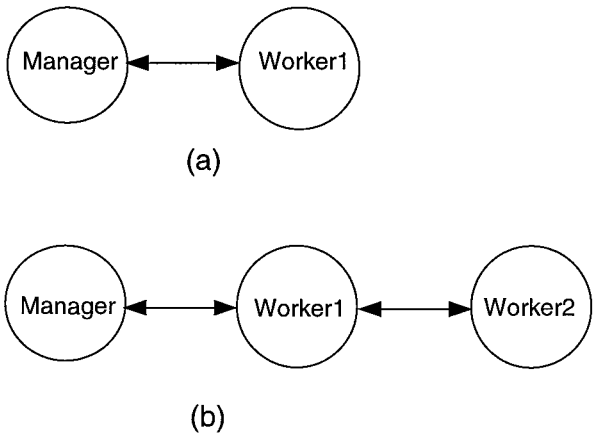


Figure 5.1: Configurations for determining β_e and β_f

For the configuration shown in Figure 5.1(a), the total execution time is given by

$$T_{total} = M(T_e + \beta_e). \tag{5.1}$$

Experiments were run on this configuration with T_e = 1, 5, 10, 20 and 40 ms and a large value of M = 10000. The value of β_e was obtained by substituting the measured

execution time in equation (5.1) for each of the five cases. As expected, for different T_e's, β_e remained constant, varying by less than 3μs. The average value of β_e was 482 μs.

For the configuration shown in Figure 5.1(b), we can express the total execution time in terms of the number of tasks processed (M_1) and forwarded (M_2) by Worker1. Worker1 spends $(T_e + \beta_e)$ for every task it processes, and β_f for every task it forwards. Thus,

$$T_{total} = M_1(T_e + \beta_e) + M_2\beta_f. \tag{5.2}$$

The same set of experiments were run on configuration 5.1(b). We measured T_{total}, M_1 and M_2 and solved for β_f by substituting these values into equation (5.2). Again, the variation between the values obtained for β_f for different cases was within 3μs. The average value of β_f was 453 μs.

The same set of experiments were run with the Trollius version of *Pfarm*, using physical layer communication. For this case, β_e was 570 μs and β_f was 1.3 ms. Even though the design is the same for both Logical Systems and Trollius versions of *Pfarm*, the implementations are slightly different. The overheads are higher in Trollius because of the higher costs of memory allocation and deallocation, and process management. For the validation experiments, the Logical Systems version of *Pfarm* was used.

5.2 Arbitrary Topologies

The analytical framework described in Section 4.2.1 was used to determine the performance of *Pfarm* on arbitrary topologies. To test the general model, we conducted several experiments on three different breadth-first spanning trees of the 8×3 and 8×8 mesh topologies.

Table 5.1 shows the predicted and measured total execution time for the different breadth first spanning trees (shown in Figure 5.2) of an 8×3 mesh. For each case, the total number of tasks is 10000 and time is given in seconds. Note that because of the large number of tasks, steady-state time dominates the execution time, so the experiments are generally testing the accuracy of the steady-state model. Although the distribution of tasks to processors is different for different breadth first spanning trees,

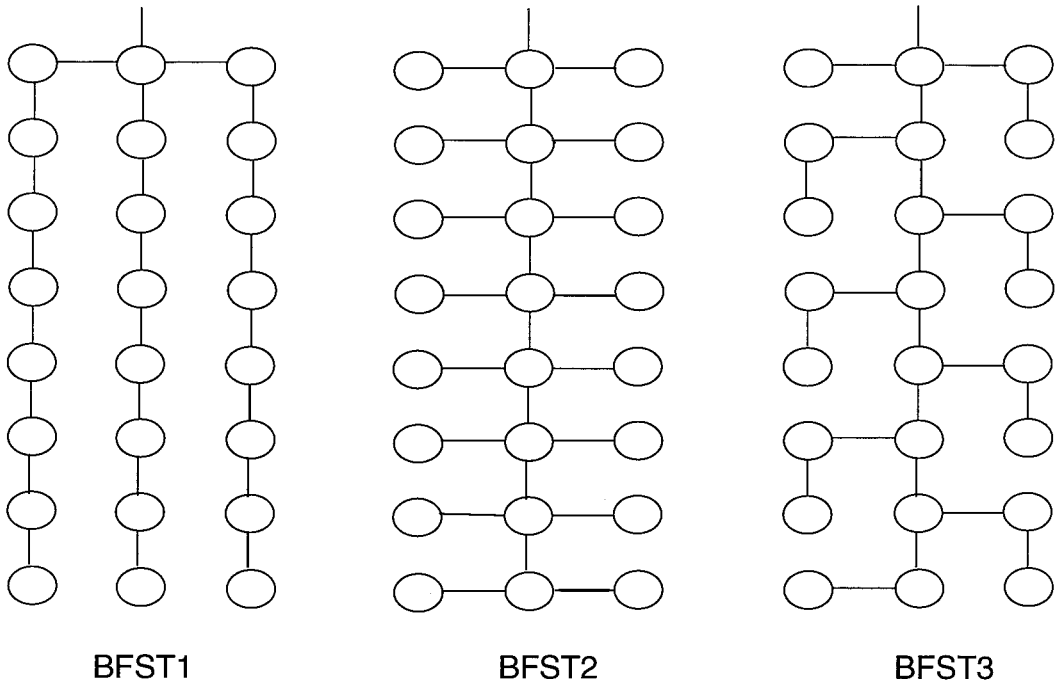


Figure 5.2: Three breadth-first spanning trees of the 8×3 mesh.

as predicted by Theorem 2, the overall execution time remains the same (neglecting the very small experimental errors).

Results for two different breadth-first spanning trees of the 8×8 mesh topology (similar to the first two BFSTs of 8×3 shown in Figure 5.2) are given in Table 5.2. Again, the total number of tasks used is 10000 and time is in seconds. As Tables 5.1 and 5.2 show, the maximum error between the predicted and measured total execution time is less than 1.5%.

For comparison purposes, in Tables 5.1 and 5.2 we have included the results of using a chain rather than a breadth first spanning tree. Note that the speedup obtained by using the spanning trees is significantly higher than that obtained by using the chain.

The minimum value of T_e used in the experiments is 10 and 30 ms for the 8×3 and 8×8 mesh, respectively. For smaller T_e, system throughput reaches the second communication bound given by equation (4.26). This scenario can be easily identified by using the analytical technique described in Section 4.2.1. If, in solving the equations,

	Breadth First Spanning Trees					Chain	
	Predicted		Measured Time			Measured	
T_e	Speedup	Time	BFST1	BFST2	BFST3	Time	Speedup
0.01	18.99	5.288	5.273	5.265	5.272	6.956	14.37
0.02	21.22	9.481	9.475	9.423	9.422	11.038	18.12
0.03	22.03	13.736	13.671	13.619	13.620	15.232	19.70
0.04	22.47	17.918	17.870	17.798	17.797	19.455	20.56
0.05	22.58	22.141	22.081	21.990	21.989	23.702	21.10
0.06	22.76	26.365	26.332	26.161	26.220	27.911	21.50
0.07	22.88	30.590	30.544	30.342	30.411	32.100	21.81
0.08	22.98	34.816	34.796	34.564	34.642	36.411	21.97

Table 5.1: Comparison of predicted and measured results for 8×3 mesh.

	Breadth First Spanning Trees				Chain	
	Predicted		Measured Time		Measured	
T_e	Speedup	Time	BFST1	BFST2	Time	Speedup
0.03	54.98	5.457	5.398	5.372	7.631	39.31
0.04	56.46	7.085	6.982	6.949	9.231	43.33
0.05	57.37	8.715	8.602	8.589	10.846	46.10
0.06	57.99	10.346	10.275	10.220	12.503	47.99
0.07	58.45	11.977	11.863	11.760	14.212	49.25
0.08	58.78	13.609	13.416	13.340	15.820	50.57
0.09	59.05	15.241	15.077	14.991	17.419	51.67
0.10	59.27	16.873	16.738	16.642	19.139	52.25

Table 5.2: Comparison of predicted and measured results for 8×8 mesh.

a processor executes a negative number of tasks, the system is communication bound. In this case, it is better to use a smaller topology. An optimal subtree can be found by removing, one by one, the leaf nodes that are farthest from the root and solving the system of equations until a feasible solution is obtained (i.e., all processors execute a positive number of tasks).

The models for start-up and wind-down were validated by running a separate set of experiments on both of the configurations. The start-up model was validated by running a number of experiments with different T_e and M and observing the task number of the first task processed by each processor. In all the cases, the task number of the first task executed at a node was same as that predicted by the model given in Section 4.2.1. The wind-down model was validated by running experiments with $M = 4N$ and observing the number of tasks executed by the leaf node in the longest path. For all the cases, this leaf node executed as many or fewer tasks compared to that predicted by the wind-down analysis given in Section 4.2.1. These experiments were performed with a varied T_e on both the 8×3 and 8×8 mesh.

5.3 Balanced Tree Topologies

In this section, we validate the performance models derived in Section 4.2.2 for balanced tree topologies. Table 5.3 gives the range of experiments conducted to validate the steady-state, start-up and wind-down models.

Model	Topology	N	M	$T_e(ms)$
Steady-state	Chain	1 to 64	10000	1,5,10,20,40
	Binary tree	1 to 63	100000	
	Ternary tree	1 to 40		
Start-up and wind-down	Chain	1 to 64	4N	1,5,10,20,40
	Binary tree	1 to 63		
	Ternary tree	1 to 40		

Table 5.3: Range of processor farm experiments

To validate the steady-state model, experiments were run using a large M so that,

in comparison start-up and wind-down was negligible. The minimum value of T_e chosen for these experiments is 1 ms because in order for *Pfarm* to make use of more than one worker, T_e must be greater than β_f, where $\beta_f = 453\mu$s. We validated the start-up and wind-down analysis separately by performing experiments with $M = 4N$. For this value of M, the total execution time consists of only the start-up and wind-down phases since the system never reaches steady state.

5.3.1 Steady-state Validation

First, we present the results of the validation experiments in which data and result sizes were small, and $(T_c + \beta_c) < \beta_f$. This ensured that the communication bound given in equation (4.25) was not reached for any of the experiments. Table 5.4 shows the percentage error between the predicted and measured execution time for a linear chain configuration. Table 5.4 shows that the percentage errors are within 3%. Also, for a fixed T_e, the total execution time continues to decrease up to a certain value of N. After this point, there is no considerable decrease in the execution time as the throughput approaches the asymptotic communication limit of $1/\beta_f$. For example, for $T_e = 1$ and 5 ms, the decrease in execution time is small after 8 and 32 nodes, respectively.

Tables 5.5 and 5.6 show the percentage error between the predicted and measured execution time for binary and ternary tree topologies, respectively. From the tables, observe that the percentage error again does not exceed 3% until the system reaches the asymptotic bound, $1/\beta_f$. Unlike in the linear chain case, the measured execution time may begun to increase as N increases after the optimal point. For example, in the ternary tree case, for $T_e = 1$ ms the optimal N is 13 corresponding to the number of nodes in a 3-level tree. However, performance degrades significantly when another processing level is added. For N larger than the optimal number, the total processing capacity of the system exceeds the rate at which tasks can flow into the system, which is $1/\beta_f$. But, any demand-driven dynamic scheduler continues to forward tasks to the workers farther down the tree as long as these workers have free buffers and there are unprocessed tasks. Thus, the workers closer to the manager execute fewer tasks since most of the time they are busy forwarding tasks. This in turn, leads to poor utilization

N	$T_e = 1$ ms			$T_e = 5$ ms		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
1	14.826	14.808	0.121	54.850	54.832	0.033
2	8.750	8.731	0.217	28.608	28.601	0.024
4	5.901	5.851	0.847	15.534	15.522	0.077
8	4.789	4.670	2.485	9.090	9.070	0.220
16	4.543	4.540	0.066	6.054	6.024	0.496
32	4.530	4.543	-0.287	4.836	4.790	0.951
48	4.530	4.566	-0.795	4.603	4.526	1.673
64	4.530	4.539	-0.199	4.548	4.588	-0.880

N	$T_e = 10$ ms			$T_e = 20$ ms		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
1	104.880	104.868	0.01	204.941	204.915	0.01
2	53.602	53.605	0.00	103.625	103.626	0.00
4	27.986	27.986	0.00	52.978	52.993	-0.03
8	15.225	15.242	-0.11	27.677	27.729	-0.19
12	11.097	11.039	0.52	19.427	19.331	0.49
16	9.020	8.965	0.61	15.235	15.173	0.41
32	6.047	6.017	0.496	9.018	9.014	0.044
48	5.186	5.177	0.174	7.019	7.056	-0.527
64	4.828	4.840	-0.249	6.068	6.123	-0.906

Table 5.4: Comparison of Predicted and Measured Total Execution Time for Processor Farm running on Linear Chain

of these workers in terms of the number of tasks locally processed, especially the root worker which only ends up processing the first task it receives. When this phenomenon occurs, the time spent by the root processor to forward all the tasks, except the first one, generally exceeds the total time taken by this processor for the cases in which the processor topology had not reached the asymptotic limit. This causes the total execution time to increase when the size of the processor topology is increased. Even though this phenomenon occurs in the linear chain case, it does not lead to any appreciable increase in the total execution time because the flow of tasks down the topology is smaller. The affect of this phenomenon for tree topologies would becomes even worse when the number of buffers on each processor was increased.

N	$T_e = 10$ ms			$T_e = 20$ ms		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
1	104.881	104.868	0.012	204.941	204.915	0.013
3	36.014	36.027	-0.036	69.369	69.370	-0.001
7	15.970	15.974	-0.025	30.261	30.267	-0.020
15	7.753	7.742	0.142	14.431	14.410	0.146
31	4.829	4.828	0.021	7.158	7.138	0.279
63	4.776	4.632	3.015	4.846	4.706	2.889

N	$T_e = 40$ ms		
	Predicted Exec Time	Measured Exec Time	% Error
1	405.061	405.029	7.900e-3
3	136.102	136.096	4.408e-3
7	58.876	58.855	0.036
15	27.820	27.788	0.115
31	13.663	13.618	0.329
63	6.864	6.820	0.641

Table 5.5: Comparison of Predicted and Measured Total Execution Time for Processor Farm running on Binary Tree.

In Tables 5.7 and 5.8 we have tabulated the percentage error between the predicted and measured total execution time on linear chain and binary tree configurations for experiments with larger data and result sizes. Both the data and result size used in these experiments are 1000 bytes per task. This leads to a larger communication time

N	$T_e = 1$ ms			$T_e = 5$ ms		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
1	14.827	14.808	0.128	54.851	54.832	0.035
4	4.811	4.809	0.042	14.628	14.633	-0.034
13	4.793	4.555	4.966	4.854	7.080	-45.859
40	4.749	6.016	-26.679	4.773	6.730	-41.001
N	$T_e = 10$ ms			$T_e = 20$ ms		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
1	104.881	104.868	0.012	204.941	204.915	0.013
4	27.116	27.133	-0.063	52.135	52.134	0.000
13	8.682	8.677	0.058	16.383	16.376	0.043
40	4.803	6.324	-31.668	5.471	5.460	0.201

Table 5.6: Comparison of Predicted and Measured Total Execution Time for Processor Farm running on Ternary Tree.

for forwarding tasks and results, and $\beta_f < (T_c + \beta_c)$. In this case, the rate of task processing will be limited by the communication latency time. The system reaches the communication bound given by equation (4.25) after an optimal value of N. This value of N depends on the computation size of tasks, T_e. From the tables, we can observe that for $T_e = 5$ ms, the system reaches its communication bound at $N = 12$ and 15 for linear chain and binary tree respectively. While the system remains in steady-state, the error is within 3%, however, once the communication bound is reached, the error is around 10% and remains almost constant as N increases. In this case, measured execution time does not increase with an increase in N. This is because it takes longer to forward a task to a child node and thus tasks do not get forwarded to the nodes farther from the manager for both chain and tree topologies. The errors obtained when the system is communication bound are larger compared to the steady-state error because of the difficulties in obtaining an accurate value for τ. The value of τ changes depending on the utilization of the link in both the directions. We have used an optimistic value for τ that leads to a slightly larger value for optimal N. This is reasonable as the performance of the system does not decrease in this case even when a larger N is used.

N	$T_e = 5$ ms		
	Predicted Exec Time	Measured Exec Time	%Error
1	54.845	55.183	-0.616
2	28.604	28.873	-0.940
4	15.532	15.795	-1.693
8	9.092	9.341	-2.739
12	7.464	8.327	-11.562
16	7.464	8.320	-11.468
24	7.464	8.312	-11.361
32	7.464	8.301	-11.214

Table 5.7: Comparison of Predicted and Measured Total Execution Time for Processor Farm running on Linear Chain under Communication Bound

N	$T_e = 5$ ms		
	Predicted Exec Time	Measured Exec Time	%Error
1	54.845	55.183	-0.616
3	19.347	19.585	-1.230
7	8.844	9.009	-1.865
15	7.464	8.172	-9.485
31	7.464	8.169	-9.445

Table 5.8: Comparison of Predicted and Measured Total Execution Time for Processor Farm running on Binary Tree under Communication Bound

5.3.2 Start-up and Wind-down Validation

Table 5.9 shows the percentage error between the predicted and measured total execution time for start-up and wind-down experiments. The maximum error observed in these experiments is approximately 15%. As explained in the start-up and wind-down analysis in Section 4.2.1, the start-up and wind-down costs obtained are upper bounds and thus the errors are larger compared to the steady-state case.

N	$T_e = 10$ ms			$T_e = 40$ ms		
	Upper Bound Exec Time	Measured Exec Time	% Error	Upper Bound Exec Time	Measured Exec Time	%Error
4	0.097	0.086	11.340	0.367	0.326	11.172
8	0.131	0.109	16.794	0.492	0.409	16.870
16	0.159	0.145	8.805	0.579	0.535	7.599
32	0.202	0.186	7.921	0.712	0.626	12.079
48	0.225	0.216	4.000	0.765	0.676	11.634
64	0.268	0.257	4.104	0.818	0.726	11.247

N	$T_e = 10$ ms			$T_e = 40$ ms		
	Upper Bound Exec Time	Measured Exec Time	% Error	Upper Bound Exec Time	Measured Exec Time	%Error
3	0.064	0.054	15.625	0.244	0.204	16.393
7	0.066	0.065	1.515	0.246	0.245	0.407
15	0.078	0.067	14.103	0.289	0.247	14.533
31	0.083	0.068	18.072	0.293	0.251	14.334
63	0.092	0.080	13.043	0.302	0.256	15.232

N	$T_e = 10$ ms			$T_e = 40$ ms		
	Upper Bound Exec Time	Measured Exec Time	% Error	Upper Bound Exec Time	Measured Exec Time	%Error
4	0.064	0.054	15.625	0.244	0.204	16.393
13	0.067	0.056	16.418	0.247	0.206	16.599
40	0.075	0.072	4.000	0.255	0.248	2.745

Table 5.9: Comparison of Predicted and Measured Total Execution Time (Start-up & Wind-down) for Processor Farm running on Linear Chain, Binary Tree and Ternary Tree.

N	$T_e = 10$ ms			$T_e = 20$ ms		
	Predicted Exec Time	Constant %Error	Uniform %Error	Predicted Exec Time	Constant %Error	Uniform %Error
1	104.880	0.010	-0.400	204.941	0.013	-0.453
2	53.602	0.000	-0.396	103.625	0.000	-0.458
4	27.986	0.000	-0.382	52.978	-0.028	-0.525
8	15.225	-0.110	-0.512	27.677	-0.188	-0.755
16	9.020	0.610	0.455	15.235	0.407	-0.217
32	6.047	0.496	-0.050	9.018	0.044	0.011
48	5.186	0.174	-0.116	7.019	-0.527	-1.225
64	4.828	-0.249	1.263	6.068	-0.906	-1.269

Table 5.10: Comparison of Predicted and Measured Total Execution Time for uniform task distribution for Processor Farm running on Linear Chain

5.4 Robustness

In all the experiments discussed so far, we have used a constant value for T_e. However, in practice, T_e may vary from one task to another. In order to test the robustness of using average values for prediction under this condition, we experimented with two common distributions for task sizes: uniform and bimodal. Experimental results are compared with those predicted by the model using the average value for T_e.

We ran several sets of experiments with uniform distribution of task sizes. For all the experiments, the total number of tasks used was 10,000. Table 5.10 shows the percentage error between the predicted and measured total execution time for two sets of experiments on a linear chain configuration. The task execution time varies from 1 to 19 ms (average $T_e = 10$ ms) for the first set, and from 1 to 40 ms (average $T_e = 20$ ms) for the second set. As Table 5.10 shows, the errors are all within 3%.

Several sets of experiments were conducted with bimodal distribution of task sizes. In these experiments M was 10000, and the values of T_e were 1, 5, 10, 20 and 40 ms. Here, we describe a set of experiments in which we used 5,000 tasks of 1 ms duration and another 5,000 of 20 ms duration. In the bimodal distribution case, the order of arrival of the tasks into the system also affects the performance. Experiments were conducted with four different arrival patterns:

1. Both 1 ms and 20 ms tasks arrive with equal probability.
2. 20 ms tasks arrive at a probability of 0.75, until all 5,000 of them are processed.
3. 1 ms tasks arrive at a probability of 0.75, until all 5,000 of them are processed.
4. All the 1 ms tasks arrive before any 20 ms tasks.

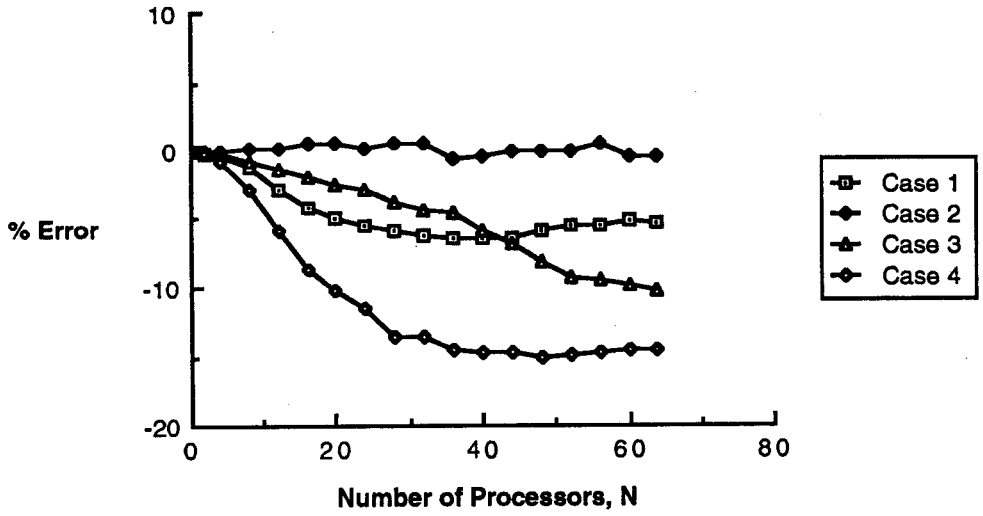


Figure 5.3: Error graph for processor farm on linear chain with tasks of bimodal distribution

In Figure 5.3, we have plotted the percentage error between the predicted and measured total execution times for these experiments on a linear chain topology. For prediction, we have used an average value of $T_e = (1 \times 5000 + 20 \times 5000)/10000 = 10.5$ ms in the model. As we can observe from the figure, the errors are within 3% for all the four cases, when N is less than 8. For larger N, the prediction is accurate for the first case but the error increases for the other three cases. The maximum error observed varies from around 6.5% in the second case to around 10.25% in the third case, and is highest at around 15.0% for the fourth case.

The errors depend on the extent to which the average T_e reflects the actual computation requirements of the tasks. As long as N is smaller than the optimal value corresponding to the smaller T_e of the two, the average value works well for all the cases.

However, for larger values of N, the average value works well only when the two kinds of tasks are well mixed with respect to arrival order, as in the first case with equal probability. Among the other three cases, tasks are mixed for a larger portion of the total execution time in the second case compared to the third case, and there is no mix at all in the fourth case. In these cases, there is a corresponding increase in the error, with the largest errors occurring for the fourth case. Obviously the average value is not appropriate for the fourth case since there is no task mixing at all. One can view the execution as occurring in two distinct phases of computation, the first consisting of all the 1 ms tasks and the second with all the 20 ms tasks. For this case, it is better to use the model twice, predicting the execution time for 1 and 20 ms tasks separately and adding them together for the total time.

There is a theoretical possibility of finding a distribution of tasks with a particular arrival pattern that could lead to arbitrarily poor performance. This is due mainly to the possible failure of the task scheduling strategy to balance the load among all the processors. However, it is very difficult to come up with this distribution as the system is dynamic and events happen in a nondeterministic order. We believe it to be unlikely for any application program to consist of tasks with such a distribution.

5.5 Chapter Summary

In this Chapter, we experimentally validated the performance models derived in Chapter 4 using a *Pfarm* implementation on a large transputer-based system. Experimentally we showed that on a fixed topology, the performance obtained by *Pfarm* is the same for any breadth-first spanning tree, as predicted by Theorem 2 in Chapter 4. We also discussed the experiments conducted to determine the values of the system overhead parameters, β_e and β_f.

Chapter 8 describes how *Pfarm* can be integrated into a programming environment that includes other programming tools such as a graphical interface, mapper and debugger. We describe the user interface for *Pfarm* and discuss how the models can be used for performance tuning.

Chapter 6

Divide-and-Conquer: Design and Modeling

In this chapter, we describe the design of *TrEK* (Tree Execution Kernel) that provides runtime system support for divide-and-conquer applications. *TrEK* is designed such that it can execute divide-and-conquer computations of any fixed degree and depth on any tree topology. We derive models that accurately describe the behavior and performance characteristics of *TrEK* or any similar system that satisfies the assumptions outlined.

Section 6.1 describes the design and implementation of *TrEK*. In Section 6.2, models that describe the start-up, steady-state, and wind-down phases of the computation on any tree topology are derived. We use this modeling technique to derive performance models for balanced tree topologies. We close this chapter by discussing how these models can be used in performance tuning and restructuring of application programs.

6.1 *TrEK*: Design and Implementation

As described in Section 3.4, *TrEK* assumes that there is a flow of *tree structured computations* that enter the root processor. Each of these tree structured computation corresponds to a divide-and-conquer task and tasks are assumed to have a known fixed degree and depth.

TrEK is a runtime kernel that runs on each worker node. In order to execute an application, *TrEK* has to be provided with three application dependent functions, `split`,

`join` and `compute`. The `split` function takes a task as input, splits it and outputs two or more subtasks. The `join` function takes two or more results as input, joins them and outputs a single result. Finally, the `compute` function takes a task as input, processes it and returns the result. An example of the task graph corresponding to an instance of a divide and conquer task is shown in Figure 6.1.

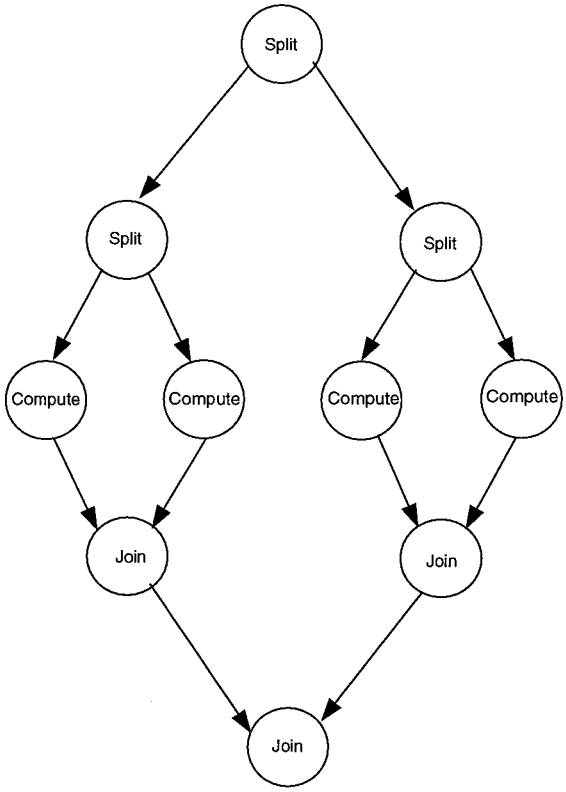


Figure 6.1: Divide-and-Conquer Task Structure

Processor farm can be viewed as a degenerate case of divide-and-conquer. Thus, it is possible to extend the *Pfarm* design to that of *TrEK*. In addition to the design issues and goals addressed by *Pfarm*, *TrEK* should also be able to execute divide-and-conquer tasks of any fixed degree and depth on any arbitrary processor topology. In this section, we explain the design modifications and extensions for *TrEK* from that of *Pfarm*.

As in *Pfarm* case, *TrEK* is designed as a set of cooperating processes in which each link is controlled by a separate process. Figure 6.2 shows the process structure of *TrEK*

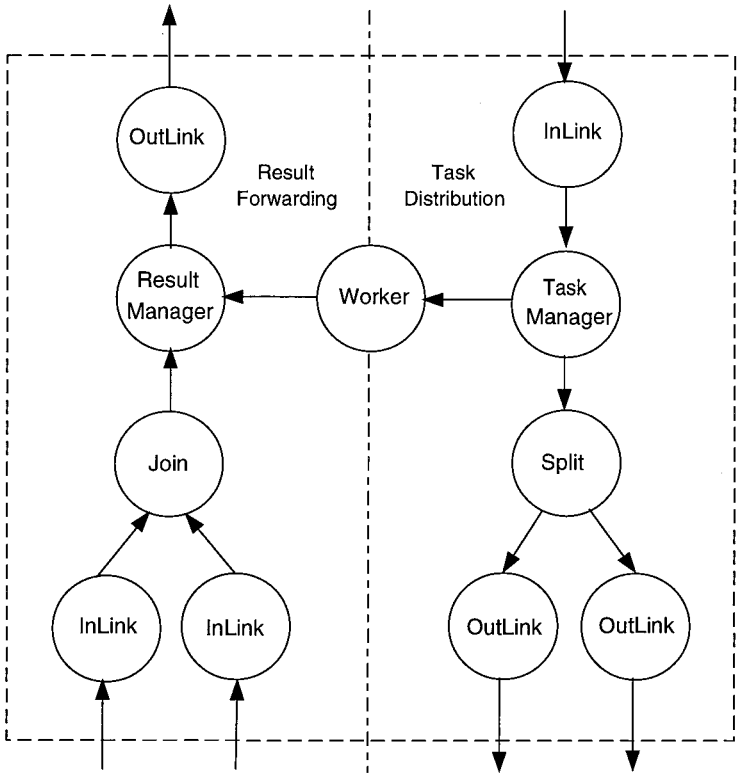


Figure 6.2: *TrEK* Process Graph on an Intermediate Worker Processor

on an intermediate worker node in the processor tree. In the task distribution path, there is an InLink process that receives tasks and a number of OutLink processes that forward subtasks to the children. In the result forwarding path, there are InLink processes that receive results of subtasks from the children, and an OutLink process that forwards results onto the parent. There is a task manager process that controls the task distribution, and there is a result manager process that controls the collection and forwarding of results. In addition to these system processes, there are three user processes on each intermediate node. The split process receives tasks from the task manager and calls the `split` function to split the tasks into two or more subtasks. The join process receives the results of subtasks from the children and calls the `join` function to combine the results. There is a local worker process on each leaf processor as well as on intermediate processors that receives tasks from the task manager and processes them

to completion.

As explained in the design of *Pfarm*, to overlap communication with computation, the communication processes and both manager processes execute at high priority, whereas the worker process executes at low priority. As the split process is in the critical path of task distribution, it must also execute at high-priority. The join process is also run at high priority as it decreases the response time which is important if there is any dependency among tasks.

In an idealized parallel implementation of divide-and-conquer algorithms on tree processors, such as those discussed in [HZ83, Col89], intermediate processors execute only split and join functions. This leads to an inefficient use of intermediate processors as they idle while waiting for the results. In *TrEK*, we allow the intermediate processors to do the processing of tasks in addition to executing split and join functions. This is possible only if the application consists of either a flow of divide-and-conquer tasks or a single divide-and-conquer task of large degree.

TrEK uses a distributed demand-driven scheduling in which children processors, whenever they have free task buffers, greedily steal subtasks from their parents. When an intermediate processor gets a new task, there are two scheduling choices. The first choice is to split the task and put the subtasks on the output queue from which children processors get their tasks. The second choice is to allocate the task for local processing. A task allocated for local processing is executed until completion by recursively solving subproblems and joining the results as in the case of uniprocessor execution of divide-and-conquer. At intermediate processors, priority is given to splitting the task and forwarding the subtasks over allocating it for local processing. The scheduling is demand driven since all the subtasks are stored in a single output queue from which the children with a free task buffer compete for tasks.

This task scheduling strategy is similar to the one used in the ZAPP [MS88] system. However, in the case of *TrEK*, there is an important difference, a processor cannot grab subtasks from its own output queue. As shown in Section 6.2, this restriction allows us to model the system and does not degrade performance.

In *TrEK*, we initially flood-fill the system with tasks. However, once full, the system is demand-driven with new tasks entering the system as the current tasks are completed. As in the case of *Pfarm*, the advantage of this scheduling strategy is that it opportunistically takes advantage of varying loads.

As in the *Pfarm* case, the number of additional task buffers to be allocated is an important design issue. On a worker processor, each link process in the task distribution path holds an active task (or subtask) and the local worker process has an active task that is being executed. Aside from these active tasks, there is an additional task buffer in the task manager for the reasons explained in the *Pfarm* case. In addition, each intermediate processor in *TrEK* has an output queue that holds the subtasks produced by the split process before forwarding them to the children processors. The output queue consists of as many buffers as the number of subtasks produced from a task. If this output queue was not present, the children processors could wait for subtasks when the parent is busy doing a split. As in *Pfarm*, we only restrict the number of task buffers, whereas we freely allocate result buffers. As results are also collected, joined and forwarded at high-priority, at any given time, the number of result buffers on a node is small.

In order to be topology independent, *TrEK* should be able to execute divide-and-conquer tasks of any degree on any topology. In *TrEK*, a parent processor does not predetermine the child to which a particular subtask is going to be forwarded. Subtasks produced by splitting a task are put in a single queue, and all the children processors get their tasks from this queue. Thus, the number of children that execute the subtasks of a particular task depends on the load, and it is possible for all the subtasks of a task to be forwarded to the same child. The Result manager on the parent node joins the appropriate subresults of a task. Therefore, it is possible to execute tasks of any degree on any topology. Each *TrEK* kernel has to be provided with the number of children of the processor on which it runs and the degree of task either at runtime or at compile time.

6.2 Performance Modeling

In this section, we derive performance models for application programs running with *TrEK* or any other system that satisfies the following assumptions. The main characteristics of the system are:

1. The hardware system is a distributed memory message passing architecture described in Section 3.2 with a linear message cost model.
2. There is a flow of fixed degree divide-and-conquer tasks into the system.
3. The depth of the tasks is equal to or greater than the depth of the underlying processor topology.
4. Tasks originate at a single source and the results are returned to the source.
5. Intermediate processors are also allowed to process the tasks in addition to executing split and join.
6. Tasks are dynamically distributed to the worker processors.

We assume that at each step, tasks are split into subtasks of equal computation. With a fixed degree divide-and-conquer task in which at each step, the work is divided into k equal parts, we have

$$W(n) = split(n) + join(n) + kW(n/k), \tag{6.1}$$

where $W(n)$ is the total amount of work of a task with an input data size of n, $split(n)$ is the work of splitting a task of size n and $join(n)$ is the work of joining the corresponding k subresults. Later in Section 7.3.5, we show experimentally that the models derived with this assumption also work well for applications in which tasks are split into subtasks of unequal computational requirements.

Our objective is to find a distribution of the load to all the processors so as to minimize the overall execution time, where load consists of both the computational requirements of the tasks and the associated overheads for splitting, forwarding and

executing them. As in the processor farm case, the system can be either computation bound or communication bound. When it is computation bound, the system acts as a pipeline with three phases to be analyzed: start-up, steady-state and wind-down. In the case of *TrEK*, start-up phase ends when all the leaf processors have received at least one subtask. At the end of the start-up phase, intermediate processors may or may not have any tasks for local processing, but they will be busy with splitting and forwarding. The definitions of steady-state and wind-down phases are same as that in the processor farm case, and the total execution time is given by,

$$T_{total} = T_{su} + T_{ss} + T_{wd} \tag{6.2}$$

First, we derive performance models for the case in which the system is computation bound. In Section 6.2.1, we present a general analytical framework to analyze the steady-state performance on arbitrary tree topologies. We also derive upper bounds for start-up and wind-down costs on arbitrary tree topologies. In Section 6.2.2, we use this analytical approach to derive models for the special case of fixed degree divide-and-conquer computations running on balanced tree topologies. We discuss the limits on performance for the communication bound case in Section 6.2.3.

6.2.1 Arbitrary Tree Topologies

Let T be a tree architecture with processors $p_1, \ldots, p_N$ And let $C(i) = \{j \mid p_j \text{ is a child of } p_i\}$ denote the children of p_i in T. Let k be the degree of the divide-and-conquer tasks to be processed.

Let $\alpha_i = T_e(i) + \beta_e$, where $T_e(i)$ is the time required for processing a subtask locally by the ith processor and β_e is the associated overhead. $T_e(i)$ is given by $W(n)$ in equation (6.1), where n is the input data size of a task that arrives at the ith processor. Let $\theta_i = T_s(i) + T_j(i) + \beta_f$, where $T_s(i)$ and $T_j(i)$ are the split and join time at the ith processor, and are given by $split(n)$ and $join(n)$ respectively. β_f is the associated overhead for every task split and forwarded to the children processors. β_f includes all the CPU overheads involved in receiving a task, splitting and forwarding the subtasks to the children processors, receiving the corresponding subresults and joining them, and

forwarding the result to the parent. Unlike in the processor farm case, β_f is not a constant because the overheads involved in forwarding subtasks and receiving results is proportional to the number of subtasks. β_f can be expressed as

$$\beta_f = \beta_{f1} + k\beta_{f2}, \tag{6.3}$$

where β_{f1} and β_{f2} are constants. β_{f1} includes the overheads required to receive a task from the parent and to send the result back, and thus, it is independent of the number of subtasks. β_{f2} is the overhead required for forwarding a subtask to a child processor and to receive the corresponding result. Thus, $\theta_i = T_s(i) + T_j(i) + \beta_{f1} + k\beta_{f2}$.

Steady-state Analysis

The steady-state phase begins once all the leaf processors have a subtask to execute and ends when the last task enters the system. It is assumed that no processor will be idle during the steady-state. Leaf processors will be busy processing the subtasks, and the intermediate processors will be busy either splitting the tasks (or subtasks), joining the results or processing the tasks (or subtasks). Suppose that M divide-and-conquer tasks of degree k are executed during this phase and let V_i denote the number of tasks (or subtasks) that visit p_i. Then, we can express steady-state execution time (T_{ss}) in terms of the number of tasks executed locally and the number of tasks forwarded along with their associated costs. T_{ss} is given by the following equation that holds for all the processors in T,

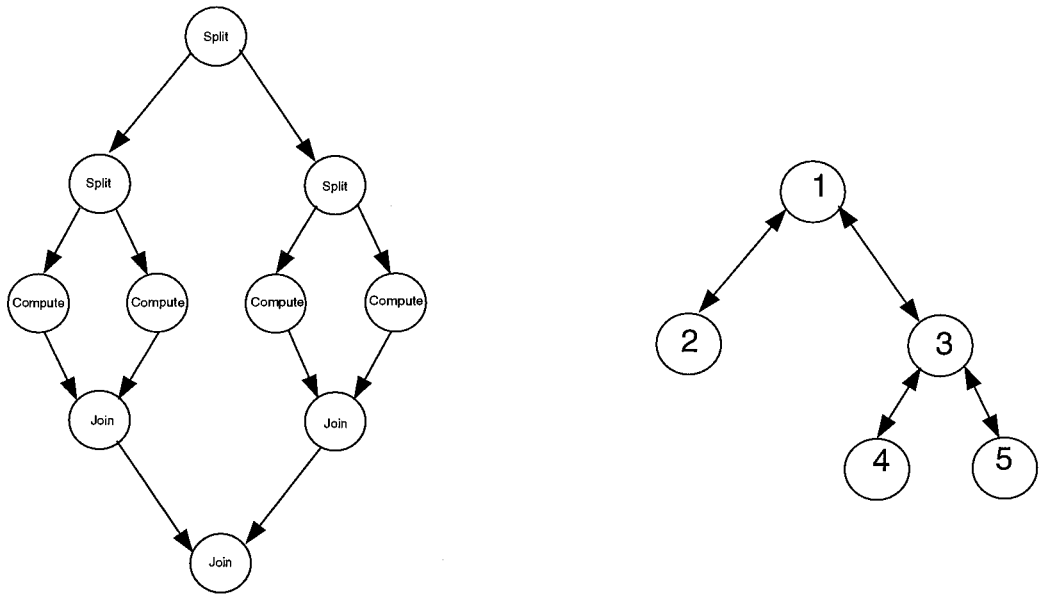
$$T_{ss} = \alpha_i(V_i - \frac{1}{k}\sum_{j\in C(i)} V_j) + \frac{1}{k}\theta_i \sum_{j\in C(i)} V_j \tag{6.4}$$

$$= \alpha_i V_i + \frac{1}{k}(\theta_i - \alpha_i)\sum_{j\in C(i)} V_j \tag{6.5}$$

The factor $1/k$ appears because of the fact that for every k subtasks that are forwarded to the children, there is only one original task, and split and join are done only once for these k subtasks.

This system of equations is similar to that of *Pfarm* and, for tree topologies, also has a unique solution. The major differences from the processor farm case are, we have α_i's

and θ_i's that vary from one processor to another unlike in the processor farm case, and the degree (k) of divide-and-conquer tasks appears in these equations. Given M, α_i's and θ_i's, we can solve for T_{ss} and V_1 to V_N. An example of this analysis is shown in Figure 6.3.



(a) Task Graph (b) Architecture Graph

$$\begin{pmatrix} \alpha_1 & \frac{1}{k}(\theta_1-\alpha_1) & \frac{1}{k}(\theta_1-\alpha_1) & & \\ & \alpha_2 & & & \\ & & \alpha_3 & \frac{1}{k}(\theta_3-\alpha_3) & \frac{1}{k}(\theta_3-\alpha_3) \\ & & & \alpha_4 & \\ & & & & \alpha_5 \end{pmatrix} \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \\ V_5 \end{pmatrix} = \begin{pmatrix} T_{ss} \\ T_{ss} \\ T_{ss} \\ T_{ss} \\ T_{ss} \end{pmatrix}$$

$$T_{ss} = \frac{M\alpha_1\alpha_2\alpha_3\alpha_4k^2}{2\alpha_1\alpha_2\alpha_3 - 2\alpha_2\alpha_3\theta_1 - 2\alpha_1\alpha_2\theta_3 + 2\alpha_2\theta_1\theta_3 + \alpha_1\alpha_2\alpha_4k + \alpha_1\alpha_3\alpha_4k - \alpha_2\alpha_4\theta_1k - \alpha_3\alpha_4\theta_1k - \alpha_2\alpha_3\alpha_4k^2}$$

Figure 6.3: An example of the steady-state analysis

This analysis gives the execution time of the steady-state phase in terms of parameters, whose values can be determined prior to the execution. The total number of tasks (M) and the degree of division (k) are usually known. $T_e(i), T_s(i)$ and $T_j(i)$ can be estimated or measured experimentally by the techniques described in Section 8.2.1.

The processor overheads β_e, β_{f1} and β_{f2} are dependent on the *TrEK* implementa-

tion and hardware processor characteristics, but are independent of the application and the hardware topology. Therefore, they need only be determined once for a particular implementation of *TrEK*.

We have experimentally found that on a fixed topology, a breadth-first spanning tree with maximum number of leaves provides maximum performance. The experiments are discussed in Section 7.2.

Start-up and Wind-down Analysis

As in the processor farm case, start-up and wind-down analysis depend on the underlying topology and the task scheduling strategy. We construct the process graph of the system by replacing each processor by the process structure given in Figure 6.2. Again, we ignore the processes that are not in the task forwarding path. The process graph of the topology given in Figure 6.4(a) is shown in Figure 6.4(b).

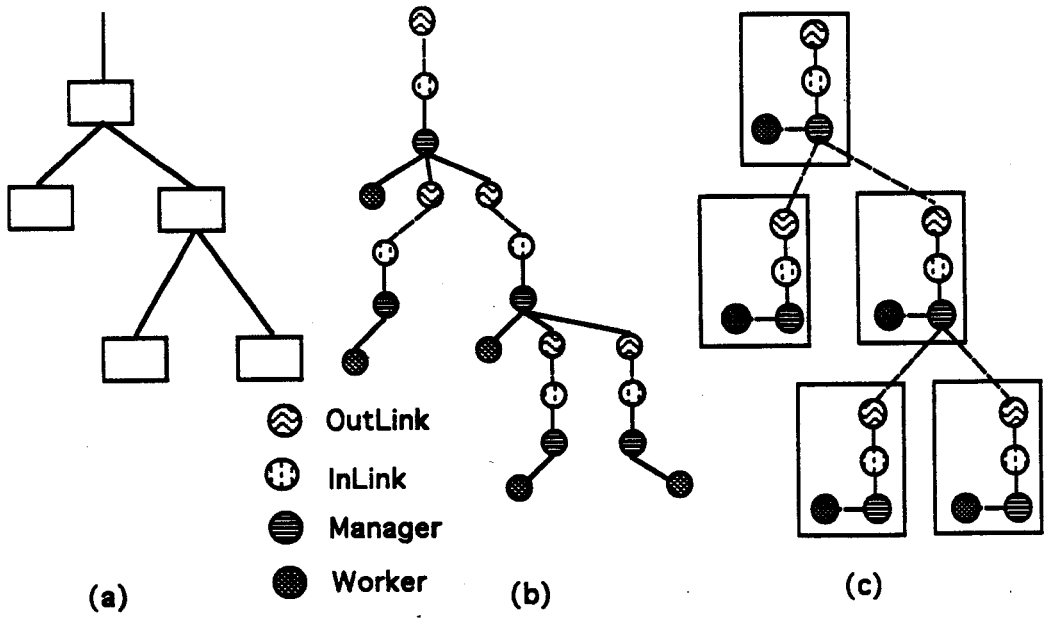


Figure 6.4: (a) node graph (b) process graph (c) subtree decomposition

Start-up and wind-down time depends on the number of task buffers in the system.

Notice that the tasks in processors at different levels are different in terms of their work. In *TrEK*, on every intermediate processor, there is one task on each of the following processes: the task receiving InLink, the task manager, and the worker. The split process produces k subtasks per task, which are put into a single output queue that can hold only k subtasks. Each task forwarding OutLink process has a single buffer to hold an outgoing subtask on that particular link. If we count this subtask towards the particular child processor to which it is getting forwarded, then every intermediate processor has five task buffers. Each leaf processor has four task buffers since the leaves do not have a split process. Thus, the total number of active tasks at any given time is given by

$$5 \sum_{i=1}^{D-1} \frac{m_i}{k^{i-1}} + 4\frac{m_D}{k^{D-1}}$$

where m_i is the number of processors at the ith level when the levels are numbered 1 to D from the root. In case of k-ary divide-and-conquer tasks running on a k-ary D level balanced tree, the total number of active tasks at any time is $5D - 1$.

Start-up

Start-up begins when the first task enters the system and ends when all the leaf processors have at least received one subtask. As explained in Section 6.1, the scheduling strategy in *TrEK* gives priority to splitting the task and forwarding the subtasks to children processors over allocating the task for local processing. When the first task enters a processor, the manager passes it on to the split process which splits the task into k subtasks that are kept in a single output queue controlled by the manager. Task forwarding OutLink processes, when free, receive a subtask from the manager process.

For analysis purposes, we use the collapsed process graph like the one shown in Figure 6.4(c). Here, we are interested in obtaining an upper bound for the start-up time. In an arbitrary tree, start-up time is given by the maximum time taken for a leaf to receive its first task. First, we will obtain an expression for the task number of the first task received by a processor. Then, we will determine the time required for a leaf processor to receive its first task. The technique used here is similar to that described in Chapter 4 for the start-up analysis for the processor farm case.

Given a rooted oriented tree T, with children nodes numbered from left to right

starting at one, let $c(v)$ be the child number of node v with respect to the parent of v, $p(v)$. Let $p^n(v)$ denote the nth ancestor of node v in T and let $\deg(v)$ be the down-degree of node v. Let k be the degree of the divide-and-conquer tasks.

Definition 2 *Let* $d_n(v)$ *equal* $\prod_{i=0}^{n} \left\lceil \frac{\deg(p^i(v))}{k} \right\rceil$.

Lemma 2 *For any rooted oriented tree* T*, the first task received by node* v *is the*

$$\left\lceil \frac{c(p^{n-1}(v))}{k} \right\rceil + \sum_{j=0}^{n-2} \left(\left\lceil \frac{c(p^j(v))}{k} \right\rceil - 1 \right) d_{n-j-2}(p^{j+2}(v))$$

task at node $p^n(v)$*, the root of* T.

Proof:

Let $s(v,i)$ be the number of the ith task received by node v. The first task to arrive at a child node v is the subtask of $\left\lceil \frac{c(v)}{k} \right\rceil$th task that arrives at the parent node $p(v)$. From then onwards, node v receives a subtask for every $\left\lceil \frac{\deg(p(v))}{k} \right\rceil$ tasks arriving at $p(v)$. Thus, we obtain the following recurrence

$$s(v,i) = s\left(p(v), \left\lceil \frac{c(v)}{k} \right\rceil + (i-1) \left\lceil \frac{\deg(p(v))}{k} \right\rceil \right).$$

In general,

$$\begin{aligned}
s(v,i) &= s\left(p^1(v), \left\lceil \frac{c(p^0(v))}{k} \right\rceil + (i-1)d_0(p(v))\right) \\
&= s\left(p(p(v)), \left\lceil \frac{c(p(v))}{k} \right\rceil + \left(\left\lceil \frac{c(p(v))}{k} \right\rceil + (i-1)d_0(p(v)) - 1 \right) d_0(p(p(v)))\right) \\
&= s\left(p^2(v), \left\lceil \frac{c(p(v))}{k} \right\rceil + \left(\left\lceil \frac{c(p(v))}{k} \right\rceil - 1 \right) d_0(p^2(v)) + (i-1)d_0(p^1(v))d_0(p^2(v))\right) \\
&= s\left(p^2(v), \left\lceil \frac{c(p^1(v))}{k} \right\rceil + \sum_{j=0}^{0} \left(\left\lceil \frac{c(p(v))}{k} \right\rceil - 1 \right) d_{2-j-2}(p^{j+2}(v)) + (i-1)d_1(p(v))\right) \\
&\vdots \\
s(v,i) &= s\left(p^n(v), \left\lceil \frac{c(p^{n-1}(v))}{k} \right\rceil + \sum_{j=0}^{n-2} \left(\left\lceil \frac{c(p^j(v))}{k} \right\rceil - 1 \right) d_{n-j-2}(p^{j+2}(v)) + (i-1)d_1(p(v))\right)
\end{aligned}$$

At the root, $s(v,i) = i$. Thus, when $p^n(v)$ is the root

$$s(v,1) = \left\lceil \frac{c(p^{n-1}(v))}{k} \right\rceil + \sum_{j=0}^{n-2} \left[\left\lceil \frac{c(p^j(v))}{k} \right\rceil - 1 \right] d_{n-j-2}(p^{j+2}(v)) \qquad (6.6)$$

The time required for the task $s(v,1)$ to arrive at node v is given by

$$T_{su}(v) = s(v,1)\left(T_{cd}(D) + T_s(D) + \frac{1}{2}(\beta_{f1} + k\beta_{f2})\right) + \sum_{i=1}^{n-1}\left(T_{cd}(i) + T_s(i) + \frac{1}{2}(\beta_{f1} + k\beta_{f2})\right),$$

where the first part of the equation represents the time after which the root node $p^n(v)$ sends a subtask to $p^{n-1}(v)$, and $T_{cd}(D)$ and $T_s(D)$ represent the communication time needed to receive a task and the split time respectively at the root node. The second part of the equation gives the time it takes for the node v to receive its first task after node $p^n(v)$ starts forwarding the corresponding task to its child. $T_{cd}(i)$ and $T_s(i)$ represent the communication time needed to receive a task and the split time respectively at node $p^i(v)$.

Start-up time for any arbitrary tree topology is given by

$$T_{su} = \max_{v \text{ a leaf}}\{T_{su}(v)\}$$

In an arbitrary topology, if the down-degree of every node is less than the degree of divide-and-conquer tasks (k), then $s(v,1) = 1$ for every node. For this case, start-up time is determined by the longest path in the topology and is given by

$$T_{su} = \sum_{i=1}^{n}\left[T_{cd}(i) + T_s(i) + \frac{1}{2}(\beta_{f1} + k\beta_{f2})\right], \tag{6.7}$$

where n is the length of the longest path. If the topology has nodes with down-degree greater than k, then $s(v,1)$ has to be evaluated for every leaf node to calculate the start-up cost. For this case, $s(v,1)$ is proportional to the number of buffers present on each node. If the topology is an unbalanced one, start-up time increases as the number of buffers increases as in the case of *Pfarm*. Start-up costs for balanced tree topologies are discussed in the next section.

Wind-down

The wind-down phase begins when the last task enters the system and ends when the last result reaches the manager. In comparison to *Pfarm*, the wind-down analysis of *TrEK* is complicated by the fact that the computation requirements of tasks at different levels are different. Tasks at the root processor have maximum computational requirements.

Here, we derive an expression for the wind-down time for an arbitrary topology using an estimate for the number of tasks executed by the root processor. In the following section, we derive an upper bound for the more interesting case, k-ary divide-and-conquer tasks on k-ary balanced tree topology.

The total number of tasks at the beginning of the wind-down phase in a tree architecture is given by the number of active tasks at any given time. As derived in the beginning of this section, it is given by

$$M_{wd} = 5 \sum_{i=1}^{D-1} \frac{m_i}{k^{i-1}} + 4\frac{m_D}{k^{D-1}} \tag{6.8}$$

where D is the number of levels in the topology and m_i is the total number of processors at ith level.

Assume that the processors are all identical and that they all do the same amount of work. Also, we neglect the overhead in forwarding tasks. Given these assumptions, an estimate on the number of tasks executed by the root processor is given by

$$M_1 = \left\lceil \frac{M_{wd}}{N} \right\rceil$$

where N is the total number of processors. In any tree architecture, the value of this varies from 1 to 4 based on the number of processors.

Thus, an estimate for the wind-down time is given by

$$T_{wd} = M_1(T_e(D) + \beta_e)$$

6.2.2 Balanced Tree Topologies

In this section, we analyze the performance of balanced divide and conquer computations on balanced tree topologies using the general framework. We hypothesize that a g-ary balanced tree topology (where g is the number of links on each node) achieves optimal performance for divide-and-conquer applications for the following reasons:

1. Experimentally, we have found that on a fixed topology, a breadth-first spanning tree with maximum number of leaves provides maximum steady-state performance.

As a g-ary balanced tree is a breadth-first spanning tree with maximum number of leaf nodes among all the topologies with the same number of nodes, it provides maximum steady-state performance.

2. As explained in the previous section, start-up cost is proportional to the length of the longest path in the topology. Balanced tree topologies have minimum length longest path among all topologies with the same number of nodes. As a result, by the analysis given in Section 6.2.1, it also minimizes start-up time.

3. Wind-down cost is also proportional to the length of the longest path in the topology. Once again, the minimal path length and the symmetry of the balanced tree also minimizes wind-down time.

First, we analyze the steady-state performance of divide-and-conquer tasks of any degree and depth on balanced tree topologies of any degree and depth. Then, we derive corresponding start-up and wind-down costs using the analyses given in the previous section for arbitrary topologies.

Steady-state Analysis

Consider a flow of l level k-ary divide-and-conquer tasks. Let g be the degree and D the number of levels of the balanced tree topology. Notice that the levels are numbered from 1 to D starting from the *leaves* rather than the root since this simplifies the derivation of the recurrence formula.

Assuming that there is no idle time, steady state time (T_{ss}) can be expressed in terms of the number of tasks processed and the number of tasks split and forwarded along with their associated costs and overheads. From the general framework, the steady-state execution time is given by equation (6.4), which is reproduced below.

$$T_{ss} = \alpha_i(V_i - \frac{1}{k}\sum_{j\in C(i)} V_j) + \theta_i\frac{1}{k}\sum_{j\in C(i)} V_j, \tag{6.9}$$

where V_i is the number of tasks (or subtasks) that visit a processor at the ith level, and $\alpha_i = T_e(i) + \beta_e$ and $\theta_i = T_s(i) + T_j(i) + \beta_{f1} + k\beta_{f2}$.

Let M be the number of divide-and-conquer tasks processed during the steady-state phase. Let f_i represent the fraction of the total number of tasks (M) processed by all

the processors at the ith level (assuming that the corresponding splits and the joins for these tasks are executed by the nodes from levels $i+1$ to D). Then, the number of subtasks processed by a processor at the ith level is given by

$$\left(\frac{k^{D-i}}{g^{D-i}}\right) f_i M,$$

since there are g^{D-i} processors at the ith level, and they have to execute k^{D-i} subtasks to finish a single original task. The number of tasks forwarded by a processor at the ith level is given by

$$\left(\frac{k^{D-i}}{g^{D-i}}\right) \sum_{j=1}^{i-1} f_j M.$$

By substituting the above in equation (6.9),

$$T_{ss} = \left(\frac{k^{D-i}}{g^{D-i}}\right) f_i M \alpha_i + \left(\frac{k^{D-i}}{g^{D-i}}\right) \left(\sum_{j=1}^{i-1} f_j\right) M \theta_i.$$

Let

$$F_i = \sum_{j=1}^{i} f_j,$$

where F_i represents the fraction of the total number of original tasks (i.e., tasks entering at the root) executed by all the processors in levels 1 through i.

Rewriting T_{ss} in terms of F_i and F_{i-1},

$$\begin{aligned} T_{ss} &= \left(\frac{k^{D-i}}{g^{D-i}}\right) (F_i - F_{i-1}) M \alpha_i + \left(\frac{k^{D-i}}{g^{D-i}}\right) F_{i-1} M \theta_i \\ &= \left(\frac{k^{D-i}}{g^{D-i}}\right) F_i M \alpha_i + \left(\frac{k^{D-i}}{g^{D-i}}\right) F_{i-1} M (\theta_i - \alpha_i) \end{aligned}$$

By rearranging the above,

$$F_i = F_{i-1} \left(\frac{\alpha_i - \theta_i}{\alpha_i}\right) + \frac{T_{ss}}{M} \left(\frac{1}{\left(\frac{k^{D-i}}{g^{D-i}}\right) \alpha_i}\right). \tag{6.10}$$

Let

$$S_i = \frac{M F_i}{T_{ss}}, \tag{6.11}$$

where S_i represents the throughput of a subtree consisting of the processors from levels 1 to i (again assuming that the corresponding splits and joins were executed at levels $i+1$ to d), and $S_0 = 0$. By substituting (6.11) into (6.10) and solving for S_i, we obtain,

$$S_i = S_{i-1}\left(\frac{\alpha_i - \theta_i}{\alpha_i}\right) + \left(\frac{1}{\left(\frac{k^{D-i}}{g^{D-i}}\right)\alpha_i}\right). \tag{6.12}$$

We are unable to obtain a closed form solution to this recurrence. Therefore, steady-state throughput of a D level balanced tree (S_D) is obtained by recursively evaluating the S_i's up to level D. In evaluating the S_i's, if $S_i > S_{i+1}$, then the intermediate processors at levels $i+1$ to D can not split and forward the tasks at the rate at which the processors in levels 1 to i can process them. The throughput limit corresponding to this case is given by equation (6.18).

Once we obtain the steady-state throughput using equation (6.12), we can derive other performance metrics such as steady-state execution time and speedup. Steady-state execution time for M tasks is given by

$$T_{ss} = \frac{M}{S_D}.$$

Steady-state speedup is given by

$$SP_D = \frac{M\alpha_D}{T_{ss}} = \alpha_D S_D,$$

where α_D is the execution time plus the associated overhead for each task at the root processor.

Start-up Analysis

Start-up time for a balanced tree is derived from the analysis given in Section 6.2.1 for arbitrary tree topologies. In the case of a balanced tree topology, start-up cost is given by the time required for the last leaf (the rightmost) to receive its first task. For a D level g-ary balanced tree topology, the task number of the first task received by the last leaf node is given by equation (6.6) with $n = D-1$.

$$s(last,1) = \left\lceil \frac{g}{k} \right\rceil + \sum_{j=0}^{D-3} \left(\left\lceil \frac{g}{k} \right\rceil - 1 \right) d_{D-j-3}(p^{j+2}(last)) \tag{6.13}$$

If $g \leq k, s(last, 1) = 1$. For this case, start-up time is given by

$$T_{su} = \sum_{i=2}^{D} \left[T_{cd}(i) + T_s(i) + \frac{1}{2}(\beta_{f1} + k\beta_{f2}) \right] \tag{6.14}$$

If $g > k, s(last, 1) > 1$ and start-up time is given by

$$\begin{aligned} T_{su} &= s(last, 1) \left[T_{cd}(D) + T_s(D) + \frac{1}{2}(\beta_{f1} + k\beta_{f2}) \right] \\ &\quad + \sum_{i=2}^{D-1} \left[T_{cd}(i) + T_s(i) + \frac{1}{2}(\beta_{f1} + k\beta_{f2}) \right] \end{aligned} \tag{6.15}$$

For example, given a 6-level 4-ary tree topology executing binary divide-and-conquer tasks,

$$\begin{aligned} s(last, 1) &= 2 + \sum_{j=0}^{3} d_{3-j}(p^{j+2}(last)) \\ &= 2 + 16 + 8 + 4 + 2 \\ &= 32. \end{aligned}$$

Wind-down Analysis

We derive an upper bound on the wind-down time T_{wd} for the case of k-ary divide-and-conquer tasks executed on balanced k-ary topologies. The total number of tasks in the system at the start of the wind-down phase (M_{wd}) is $5D - 1$, where D is the number of levels of the topology.

We discretize the wind-down phase into a number of steps, where a step is the time to execute a subtask at a leaf in the tree. Note that each leaf has 4 subtasks and all the remaining processors have 5 tasks (or subtasks). An upper bound is obtained by determining the number of steps required for each processor to contain at most one task.

Let us derive the number of steps required to reduce the number of tasks at the root to one, the task being executed. Consider a D-level tree ($D > 2$), which consists of a root and two $D - 1$ level subtrees. Assume that at the end of a step, tasks at the root are split and forwarded as far as possible towards the leaves. This is an optimistic assumption since when the tasks are not transferred to the leaves, there is less overhead and the load is better balanced. In particular, at the end of the first step, once the

leaves have finished a task, it indirectly reduces the number of tasks at the root by one. At the end of the second step, two tasks at the bottom two levels finish, and only two tasks remain at the root. After the third step, the root contains only one task (the task being executed). Thus, after three steps, there are k subtrees each with a maximum of $(5(D-1)-1)/k$ tasks (this is an upper bound on the total number of tasks as some tasks at intermediate levels in the tree have been partially processed). Note that the root is still executing a task.

By recursively applying the above argument to the roots of each of the k subtrees, after $3(D-2)$ steps, there remain only 2-level trees. In the trees that remain, the root has 5 tasks and each leaf has 4 tasks. After one step, the leaves finish one task which reduces the number of tasks at the root to 4. At the end of the second step, in addition to the leaves, the root also finishes a task reducing the total number of tasks to two. After three steps, only the leaf processors will be left with four complete tasks each. Thus, after $3D$ steps, each processor has only one task that it is executing or no task at all.

The wind-down time (T_{wd}) also depends on the time it takes to execute a single task at the root. Therefore,

$$T_{wd} \leq \max\{(3D+1)(T_e(1)+\beta_e), (T_e(D)+\beta_e)\} \tag{6.16}$$

where $T_e(1)$ is the execution time of a subtask at the leaf level and $T_e(D)$ is the execution time of a subtask at the root. For larger D ($D > 5$, small β_e), the execution time of a single task at the root dominates the wind-down time.

6.2.3 Communication Bounds

The performance models derived in Sections 6.2.1 and 6.2.2 are applicable only when the system is computation bound. In this section, we discuss the performance of the system when it is communication bound. In this case, processors may idle as the system never reaches steady-state.

There are two factors that may cause the system to be communication bound.

Case (i)

As in the processor farm case, overall performance of the system can be bound by the transfer costs whenever the data and result sizes are sufficiently large. In a divide-and-conquer task, the data and result sizes generally decrease towards the bottom of the tree. Thus, overall throughput of the system is bound by

$$S_{com1} = \frac{1}{T_c + \beta_c}, \tag{6.17}$$

where $T_c = \max\{T_{cd}(D), T_{cr}(D)\}$. As in the processor farm case, β_c is the processor overhead to receive a task from a parent or to send a result to the parent. As the link processes are identical in both *Pfarm* and *TrEK*, the value of β_c is also the same.

Case (ii)

In the second case, CPU time in transferring the tasks and results can limit the overall throughput. An intermediate processor at the ith level has to incur a CPU cost of $T_s(i) + T_j(i) + \beta_{f1} + k\beta_{f2}$ for every task that is split and forwarded. In any divide-and-conquer application, the sum of split and join costs is maximum at the root of the computation. Thus, overall throughput of the system is limited by the rate at which the root processor can split and forward the tasks independent of the number of processors. This bound is given by

$$S_{com2} = \frac{1}{T_s(D) + T_j(D) + \beta_{f1} + k\beta_{f2}}. \tag{6.18}$$

6.3 Discussion

In this section, we discuss how the performance models derived in Section 6.2 can be used for performance tuning.

6.3.1 Optimal N and Topology

Performance models can be used to determine the optimal topology and the number of nodes to be used to obtain maximum performance for a given divide-and-conquer application.

In Figure 6.6, we plotted throughput as a function In Figure 6.5, we have plotted

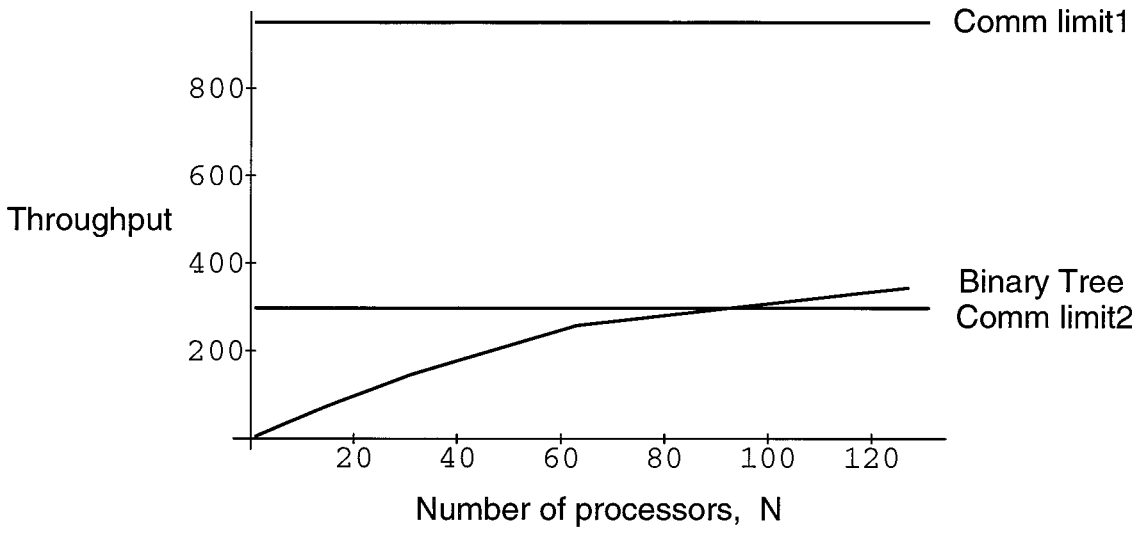


Figure 6.5: Plot of throughput curves for Binary Divide-and-Conquer Tasks on Binary Tree

the three throughput equations (6.12), (6.17) and (6.18) for a binary tree topology with a set of typical parameter values. The optimal number of processors is given by the intersection of the equations (6.12) and (6.17) or (6.18) which ever leads to the minimum throughput. Beyond this optimal value, there will be no increase in the performance with an increase in N.

In Figure 6.6, we plotted throughput as a function of N for balanced binary and ternary tree topologies executing 4-ary divide-and-conquer tasks. As mentioned in the Section 6.1, *TrEK* is topology independent and hence can be used to run any k-ary divide-and-conquer computation on any topology. As expected, for any particular N, a ternary tree topology achieves better throughput than the binary tree case. In general, it is always better to use a g-ary tree topology, where g is the maximum number of links available on each node. If the number of nodes available is less than the optimal for a given application, and it does not lead to a complete g-ary tree, it is better to use a topology in which all the levels, except the last one, are complete g-ary tree and the remaining nodes are balanced in the last level.

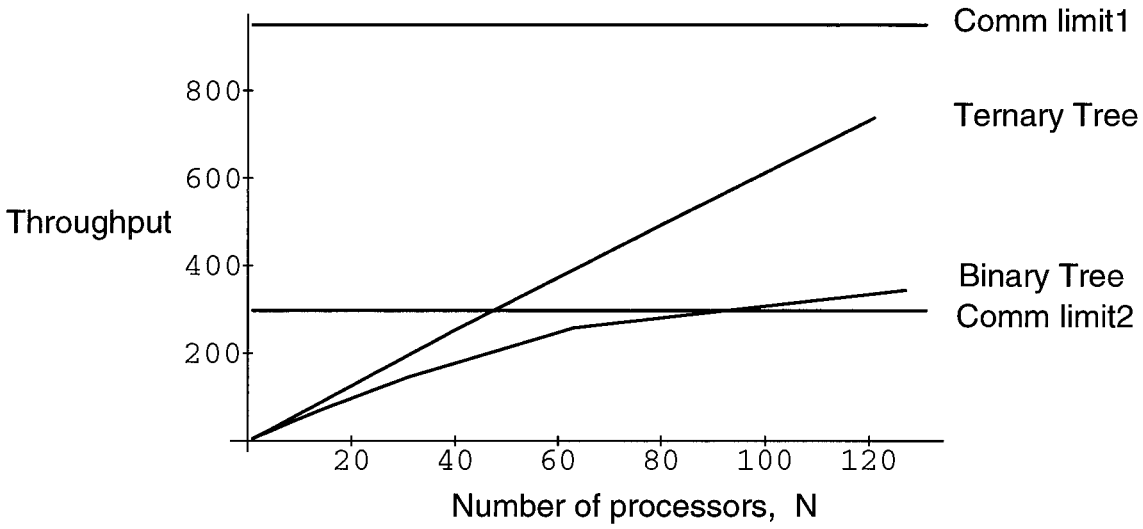


Figure 6.6: Comparison of divide-and-conquer throughput on binary tree and ternary tree topologies

As in *Pfarm case*, if the number of nodes available is larger than the optimal number of nodes to be used for a given application, one case use multiple g-ary trees to increase the overall performance.

6.3.2 Problem Scaling

As in the case of *Pfarm*, the most effective way to scale the problem is by increasing the granularity of the tasks. In Figure 6.7, we have plotted the steady-state speedup for a binary tree topology for different values of $T_e(D)$. As shown in Figure 6.7, the steady-state speedup increases with increasing values of $T_e(D)$ as long as the system does not reach any of the two communication bounds.

6.4 Chapter Summary

In this chapter, we have described the design of *TrEK*, a runtime kernel for executing divide-and-conquer applications. We described how the *Pfarm* design was modified and

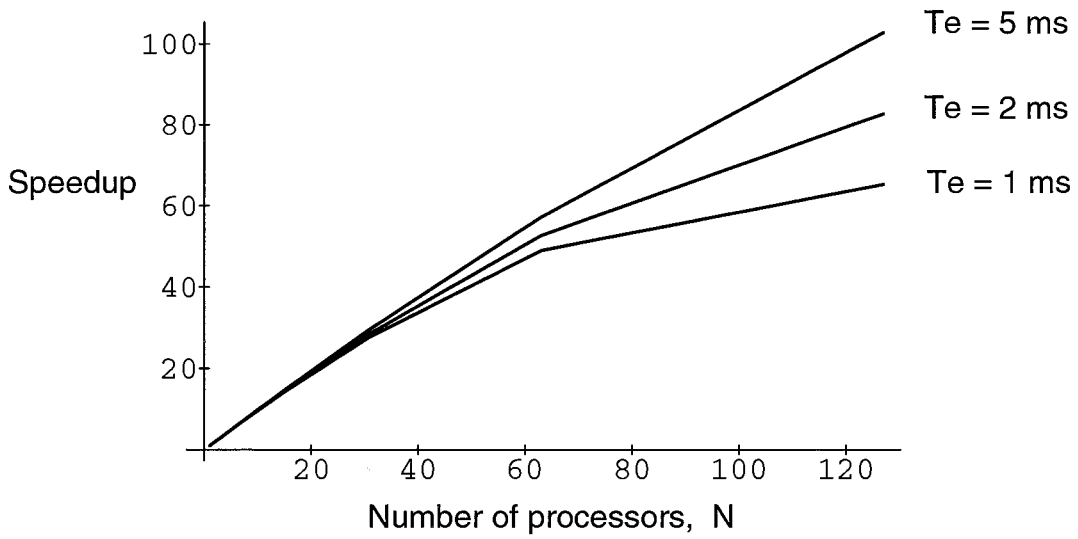


Figure 6.7: Measured speedup for divide-and-conquer on binary tree

extended for *TrEK*. We developed a general analytical framework that can be used to analyze performance of divide-and-conquer applications using *TrEK*. This framework was used to derive performance models for fixed degree divide-and-conquer problem on balanced tree topologies. In the next chapter we describe our experimental results.

Chapter 7

Divide-and-Conquer: Experiments

The performance models derived in Chapter 6 for divide-and-conquer applications were experimentally validated using *TrEK* implemented in C on Logical Systems environment.

The application program used in the validation experiments consists of a set of divide-and-conquer tasks with a synthetic workload. As in the *Pfarm* case, the application program executes empty loops corresponding to split, join and compute functions. The number of iterations of the empty loops determine the values of T_s, T_j and T_e used in a particular experiment. In this chapter, each divide-and-conquer task is represented by the following parameters: degree (k), number of levels (l), base case computation time (T_e), split time ($T_s(i)$) and join time ($T_j(i)$). The base case computation time represents the time needed for solving a leaf subtask of a divide-and-conquer task.

The experiments for determining the system overhead parameters are described in Section 7.1. In Section 7.2, we validate the performance models for arbitrary tree topologies. Performance models for balanced tree topologies are validated in Section 7.3. It is often possible to execute a divide-and-conquer application using processor farm paradigm. In Section 7.4, we compare the performance of *Pfarm* and *TrEK*.

7.1 Determining System Overheads

For experimental validation purposes, it is necessary to determine the values of the system overhead parameters, β_e and β_f. As explained in Section 6.2, β_e is the processor overhead to execute a task locally and β_f is the processor overhead for every task that is split and forwarded to the children, its value depends on the degree of the divide-and-conquer tasks. As defined in Chapter 6, $\beta_f = \beta_{f1} + k\beta_{f2}$, where k is the degree of divide-and-conquer tasks. The overhead parameters, β_e, β_{f1} and β_{f2} are constants as they correspond to the software costs to execute particular parts of the *TrEK* program. Also, these values do not depend on the topology being used. This is due to the fact that subtasks are put into a single output queue. As a result, the overhead to forward a task is a function of the cost of adding and deleting from a queue, both constant time operations. In addition, values of these overheads do not depend on the characteristics of the application program. However, they are dependent on the implementation of *TrEK*, and the underlying processor characteristics. Thus, values of these parameters have to be determined only once for a particular implementation of *TrEK*. Once these values have been determined, it is possible to predict the performance of an application that fits the model.

The overheads β_e, β_{f1} and β_{f2} are determined by conducting several experiments on simple configurations shown in Figure 7.1.

The value of β_e is determined by solving for β_e in the expression

$$T_{total} = M(T_e + \beta_e), \tag{7.1}$$

where T_{total} is the execution time for the configuration shown in Figure 7.1(a). Experiments were run on configuration 7.1(a) with T_e = 5, 10, 20 and 40 ms and a large M = 10000. By using M, T_e and measured T_{total}, one can solve for β_e. As expected, for different T_e's, β_e remained constant, varying by less than 3 μs. Changing M had no effect on the value of β_e. The average value of β_e was 560 μs.

To determine the values of β_{f1} and β_{f2}, experiments were conducted on the configurations shown in Figure 7.1(b) and (c). For these configurations, total execution time can be expressed in terms of the number of tasks processed (M_1) and forwarded (M_2) by

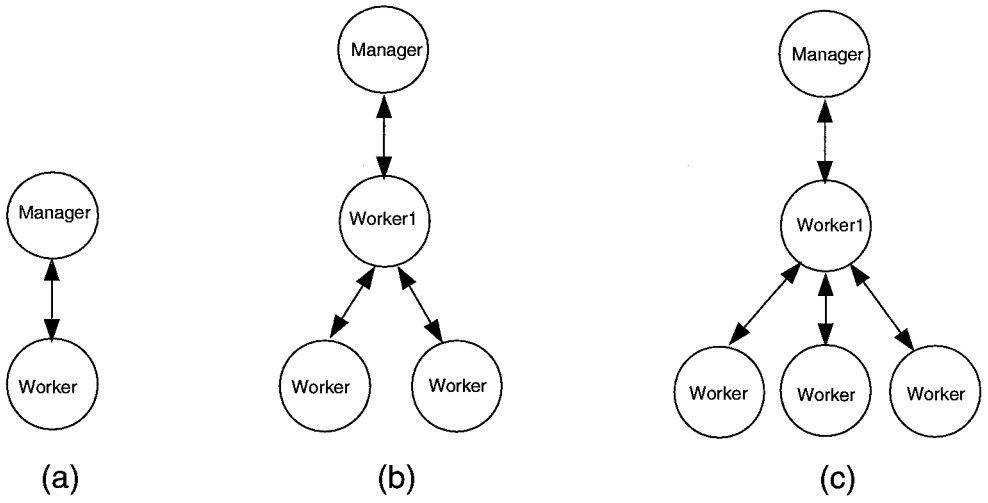


Figure 7.1: Configurations for determining β_e and β_f

Worker1. Worker1 spends $T_e+\beta_e$ for every task locally processed, and $T_s+T_j+\beta_{f1}+k\beta_{f2}$ for every task that is split and forwarded. On the configurations in Figure 7.1(b) and (c), experiments were run with divide-and-conquer tasks of degree (k) 2 and 3, respectively, with the number of levels (d) equal to 2. By choosing a sufficiently large M, we can neglect the effect of start-up and wind-down. For sufficiently large M, the total execution time for configuration in Figure 7.1(b) is given by

$$T_{total} = M_1(T_e + \beta_e) + M_2(T_s + T_j + \beta_{f1} + 2\beta_{f2}), \tag{7.2}$$

and for configuration in Figure 7.1(c), it is given by

$$T_{total} = M_1(T_e + \beta_e) + M_2(T_s + T_j + \beta_{f1} + 3\beta_{f2}). \tag{7.3}$$

For each configuration, we ran several experiments with base case T_e equal to 5, 10, 20 and 40 ms. In all experiments, the split and join time was held constant at 1 ms, and $M = 10000$. For each experiment, we measured the T_{total}, and M_1 and M_2. These values were then used in equations (7.2) and (7.3) to obtain the values of β_{f1} and β_{f2}. The variation observed between the values obtained for β_{f1} and β_{f2} for different values of T_e was within 5 μs. The average values obtained for β_{f1} and β_{f2} were 520 μs and 420 μs, respectively.

7.2 Arbitrary Topologies

In this section, we validate the general analytical framework described in Section 6.2.1 and show, experimentally, that on a fixed topology, a breadth-first spanning tree (BFST) with maximum number of leaves outperforms other BFSTs. As described in Chapter 6, by splitting a task, we can increase the amount of parallelism and make better use of the available parallelism in the hardware, but every split increases the total amount of work because of its associated overhead. In the case of a flow of divide-and-conquer tasks, ignoring start-up, it is better to reduce the number of splits since the application consists of a number of divide-and-conquer tasks. Thus, on a fixed topology, a BFST that does the minimum number of splits obtains the best performance since the overall overhead in this case is smaller compared to other BFSTs. Since splits have to occur at internal nodes, a BFST with a minimum number of internal nodes or maximum number of leaves does minimum number of splits. Experiments were conducted on three different breadth-first spanning trees (shown in Figure 7.2) of an 8×3 mesh topology.

T_e	Measured Total Execution Time		
	BFST1 (3 leaves)	BFST2 (16 leaves)	BFST3 (9 leaves)
0.001	101.675	70.922	90.889
0.002	130.038	92.097	120.634
0.003	188.199	113.763	159.948
0.004	208.821	135.700	186.049
0.005	229.898	157.455	211.225

Table 7.1: Performance Comparison of three different BFSTs of the 8×3 mesh.

Table 7.1 shows the measured execution time with the percentage error from the corresponding predicted execution time for three breadth-first spanning trees of the 8×3 mesh. In the table, times are given in seconds. For each case, a total of 1000 binary divide-and-conquer tasks with 10 levels were used. The value of T_e shown in the table is the base case computation time. A value of 1 ms was used for split and join costs at each level. Tasks of 10 levels were chosen for these experiments because the number of levels of tasks has to equal or exceed the number of levels of the topology,

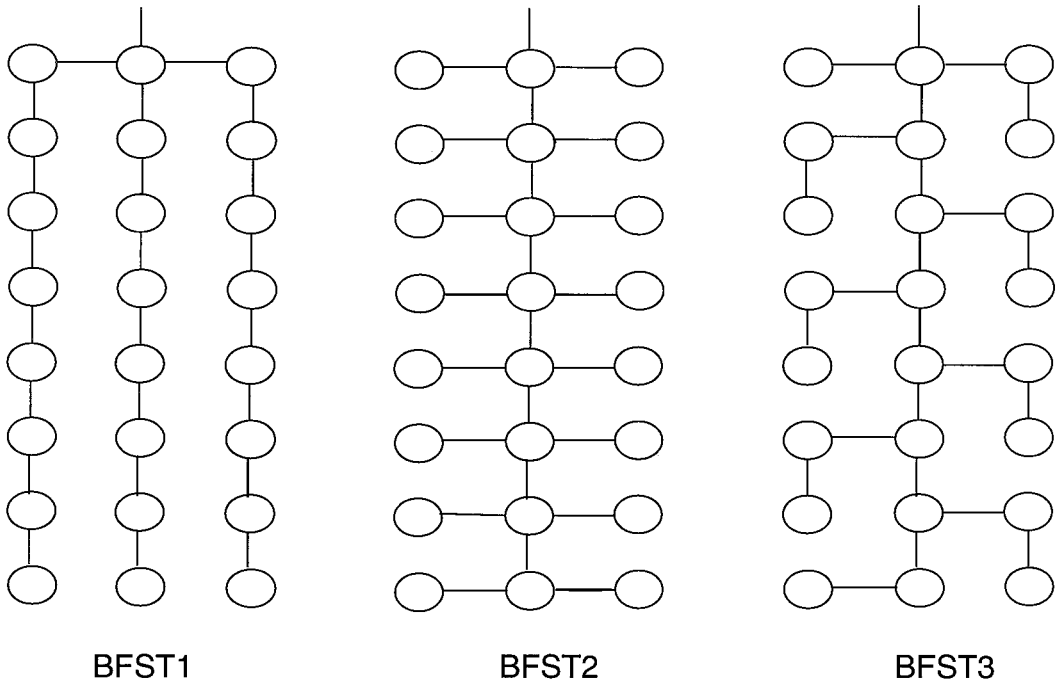


Figure 7.2: Three breadth-first spanning trees of the 8×3 mesh.

9 in this example. In order to achieve steady state, the computation time of a subtask at any node has to be greater than the sum of split and join times at the parent node plus the associated overhead β_f. Therefore, for the values of split and join times chosen in these experiments, base case T_e must be at least 1 ms. As Table 7.1 shows, the measured execution time for BFST2 is small compared to the other two BFSTs for all cases. Experimentally, this supports our claim that on a fixed topology, a BFST with maximum number of leaves provides better performance compared to other BFSTs.

7.3 Balanced Tree Topologies

In this section, we experimentally validate the performance models for *TrEK*, derived in Section 6.2.2 for balanced tree topologies. The experiments were conducted to test the models for steady-state, start-up and wind-down for k-ary divide-and-conquer tasks on g-ary balanced tree topology, variable split and join costs, and communication bound. The parameter values were chosen to satisfy the following conditions:

1. the number of levels of tasks has to be equal to or greater than the number of levels of the topology.

2. the computation time of a subtask at any node has to be greater than the sum of split and join time at the parent node plus the associated overhead β_f.

7.3.1 Steady-State

Table 7.2 gives the range of experiments conducted to validate the steady-state model. Steady state performance models were validated by experiments with a sufficiently large

Model	Topology	N	M	Tasks				
				k	l	T_e	T_s	T_j
Steady-state	Binary tree	1 to 63	10000	2	6	5,10,20	1	1
					7	1,2,5	1	1
	Ternary tree	1 to 40	10000	3	4	5,10,20	1	1
					5	1,2,5	1	1

Table 7.2: Range of Divide-and-Conquer steady-state Experiments

$M(10000)$ so that start-up and wind-down time can be ignored. For all experiments, a value of 1 ms was used for both T_s and T_j at each level. Experiments with variable split and join costs are described separately. A value of 1 ms was chosen for both split and join costs as larger values either limit the overall throughput (also discussed later) or require very large divide-and-conquer tasks to be in steady-state.

Tables 7.3 and 7.4 show the percentage difference in predicted and measured execution time for binary and ternary tree cases respectively. In these experiments, divide-and-conquer tasks of 7 and 4 levels were used on binary and ternary tree respectively. As can be observed from the tables, the errors are within 7%.

7.3.2 Start-up and Wind-down

In the case of k-ary divide-and-conquer tasks running on k-ary balanced tree topologies, there will be $5D - 1$ tasks in the system at any given time, where D is the number of levels of the hardware topology. Start-up and wind-down models were validated with

N	$T_e = 1$ ms			$T_e = 2$ ms		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
1	1906.550	1906.789	-0.013	2546.870	2547.160	-0.011
3	639.845	639.927	-0.013	853.269	853.410	-0.017
7	278.198	278.384	-0.067	369.453	369.906	-0.122
15	133.815	134.688	-0.652	176.502	179.021	-1.427
31	69.055	70.124	-1.548	88.593	90.419	-2.061
63	39.135	39.285	-0.383	48.678	49.234	-2.061

Table 7.3: Steady-state Performance Comparison for Divide-and-Conquer running on Binary Tree.

N	$T_e = 5$ ms			$T_e = 10$ ms		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
1	1616.405	1616.625	-0.013	2967.080	2967.400	-0.010
4	409.732	414.349	-1.126	747.264	747.312	-0.006
13	131.755	139.405	-5.806	235.148	247.677	-5.328
40	49.405	52.625	-6.517	82.75	87.744	-6.035

Table 7.4: Steady-state Performance Comparison for Divide-and-Conquer running on Ternary Tree.

$M = 5D - 1$ so that the system never reaches steady-state and the total execution time consists only of start-up and wind-down.

Tables 7.5 and 7.6 show the percentage error between the predicted and measured total execution time for these experiments. In these experiments, the values used for the base case computation (T_e) were 10 and 20 ms. The errors observed are large, especially for larger topologies because the predicted wind-down costs are upper bounds. We have calculated the wind-down cost based on the number of tasks that are predicted to have been executed on the root processor. For larger topologies, the model uses an upper bound of two tasks on root processor. If in the actual execution, the root processor executes only one task, then the error can be as large as 40%, because there are only a few large tasks and each task contributes considerably to the total execution time.

N	$T_e = 10$ ms			$T_e = 20$ ms		
	Upper Bound Exec Time	Measured Exec Time	% Error	Upper Bound Exec Time	Measured Exec Time	%Error
3	1.546	1.338	13.45	2.826	2.459	12.99
7	1.171	0.954	18.53	2.132	1.755	17.68
15	0.795	0.573	27.92	1.435	1.053	26.62
31	0.803	0.461	42.59	1.444	0.781	46.20
63	0.812	0.477	41.25	1.452	0.781	46.21

Table 7.5: Start-up and Wind-down Performance Comparison for Divide-and-Conquer running on Binary Tree.

N	$T_e = 10$ ms			$T_e = 20$ ms		
	Upper Bound Exec Time	Measured Exec Time	% Error	Upper Bound Exec Time	Measured Exec Time	%Error
4	1.201	0.793	33.97	1.717	1.513	11.88
13	0.913	0.635	30.45	1.163	0.695	40.24
40	0.634	0.365	42.43	1.174	0.635	45.91

Table 7.6: Start-up and Wind-down Performance Comparison for Divide-and-Conquer running on Ternary Tree.

As the divide-and-conquer tasks are generally large, it is important to validate the models for the cases in which the total number of tasks is not very large. Thus, we

conducted several experiments with $M = 1000$. Tables 7.7 and 7.8 show the percentage error between the predicted and measured total execution time for these experiments on binary and ternary tree topologies. As Tables 7.7 and 7.8 show, the errors are within 7%.

N	$T_e = 5$ ms			$T_e = 10$ ms		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
1	222.674	222.694	0.000	382.754	382.786	0.000
3	74.728	74.788	-0.080	128.013	128.325	-0.144
7	32.576	32.436	0.429	55.236	55.358	-0.221
15	15.641	15.581	0.384	26.406	26.193	0.806
31	8.165	7.911	3.110	13.487	13.086	2.973
63	4.708	4.693	0.319	7.407	7.372	0.472

Table 7.7: Comparison of Predicted and Measured Total Execution Time for Divide-and-Conquer running on Binary Tree with $M = 1000$.

N	$T_e = 1$ ms			$T_e = 5$ ms		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
1	161.640	161.662	-0.013	296.708	296.740	-0.010
4	41.053	41.003	0.122	74.883	74.758	0.167
13	13.294	13.308	-0.105	23.779	24.811	-4.340
40	4.996	4.990	0.120	8.365	8.613	-2.965

Table 7.8: Comparison of Predicted and Measured Total Execution Time for Divide-and-Conquer running on Ternary Tree with M =1000.

7.3.3 k-ary Tasks on g-ary Balanced Topologies

The experiments discussed in the previous sections tested binary and ternary divide-and-conquer tasks which exactly match the underlying topologies. In order to validate the models for cases in which the task structures does not match the underlying topologies, we conducted experiments in which binary divide-and-conquer tasks were run on ternary tree topologies. Table 7.9 shows the percentage error between the predicted and

measured total execution time for these experiments. The errors are within 7%.

N	$T_e = 10$ ms			$T_e = 20$ ms		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
4	96.351	96.292	0.061	176.630	176.436	0.110
13	30.165	31.422	-4.167	55.086	56.077	-1.780
40	10.458	10.084	3.576	18.945	19.678	-3.869

Table 7.9: Comparison of Predicted and Measured Total Execution Time for Binary Divide-and-Conquer tasks running on Ternary Tree

We claim in Chapter 6 that the models can also be used for single divide-and-conquer problem with large degree and depth. To substantiate this claim, several experiments were run, each with a single large divide-and-conquer task.

In the first example, a 5-ary 9-level divide-and-conquer task with base case computation time of 5 ms was run on a 40-node ternary tree. For prediction purposes, we evaluated the throughput excluding the root worker because in *TrEK*, with a single task, the root worker does not process any subtasks as it splits the task and forwards all the subtasks to its children. The model predicted a total execution time of 275.581 seconds whereas the *TrEK* execution took 316.478 seconds. As a second example, a 6-ary 8-level divide-and-conquer task with base case computation time of 5 ms was run on a 40-node ternary tree. For this case, the model predicted a total execution time of 232.683 ms, and the *TrEK* execution took 252.117 ms. The errors are larger than 7% because the processors closer to the manager (especially the root) can idle as the number of subtasks arriving at these processors is small.

7.3.4 Variable Split and Join Costs

In all the previous experiments, split and join costs were kept constant at all levels of the computation. We conducted several experiments to verify the validity of the models for the cases in which the split and join costs are different at different levels of computation. Results of two sets of experiments are tabulated in Tables 7.10. These experiments were conducted with 1000 binary divide-and-conquer tasks of 6 levels with base case

computation of 10 ms on a binary tree.

In the first set of experiments, a variable split cost was used where as the join cost was kept constant at all levels. The split cost was varied from 1 to 5 ms, and a value of 1 ms was used for the join cost. The second set of experiments were conducted with variable join costs (varying from 0.5 to 2.5 ms) with a constant value of 1 ms for split cost. Both the split and join costs were varied in the third set of experiments. Once again, the errors observed in these experiments are within 7%.

N	Variable Split & Constant Join			Constant Split & Variable Join		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
1	408.764	408.923	-0.038	369.745	369.889	-0.038
3	136.688	136.980	-0.217	123.679	124.019	-0.275
7	58.962	59.107	-0.246	53.381	53.587	-0.386
15	28.185	28.011	0.617	25.550	25.483	0.262
31	13.989	14.020	-0.222	12.708	12.844	-1.070
63	7.852	8.250	-5.069	6.830	7.151	-4.700

N	Variable Split & Join		
	Predicted Exec Time	Measured Exec Time	%Error
1	434.777	434.854	-0.018
3	145.363	145.736	-0.257
7	62.688	62.851	-0.260
15	30.014	29.774	0.800
31	14.939	15.129	-1.272
63	11.363	11.797	-3.819

Table 7.10: Comparison of Predicted and Measured Total Execution Time for Divide-and-Conquer Tasks with Variable Split & Join Costs

When split and join costs are large, the system performance is bound by the rate at which the root processor can split the tasks and join the results and throughput is given by equation (6.18). Several experiments were conducted to test the performance of the system for this case. Table 7.11 shows the predicted and measured execution time for a set of experiments in which 1000 binary divide-and-conquer tasks of 6 levels were run

on binary trees. The split and join costs were varied from 1 to 5 ms, with base case computation being 10 ms.

N	Variable Split & Join		
	Predicted Exec Time	Measured Exec Time	% Error
1	548.834	548.930	-0.017
3	183.388	183.769	-0.208
7	79.045	79.213	-0.216
15	37.815	37.434	1.008
31	21.370	24.534	-14.806
63	21.370	27.656	-29.415

Table 7.11: Comparison of Predicted and Measured Total Execution Time for Binary Divide-and-Conquer Under Split and Join Bound

Table 7.11 shows that the system performance reaches the limit for a 31 node binary tree topology. For larger topologies, system performance degrades as the dynamic demand-driven scheduling strategy keeps forwarding tasks towards the leaf nodes causing the nodes closer to the manager to idle. This phenomenon is similar to that observed in the processor farm case and shows the importance of determining the right number of nodes and topology to be used for a given application.

7.3.5 Robustness

In all the experiments discussed above, it is assumed that the tasks are split into subtasks that take equal amount of computational time. In general, tasks may not always be divided into equal sized tasks. To test the robustness of using the average value of T_e, we experimented with tasks which were split into k randomly unequal subtasks. Table 7.12 show the percentage error between the predicted and measured total execution time for experiments in which binary divide-and-conquer tasks were run on binary tree topologies. Two sets of experiments were conducted with tasks in which the sum of all the base case computations per task were 256 and 512 ms. It was assumed that the split and join costs are proportional to the computational requirements of the task. The percentage

errors in this case are slightly larger, but are still around 10%. The measured execution time is always larger than the predicted execution time because some of the subtasks are smaller than the overhead required to forward them, causing additional work.

N	$T_e = 256$ ms			$T_e = 512$ ms		
	Predicted Exec Time	Measured Exec Time	% Error	Predicted Exec Time	Measured Exec Time	%Error
1	297.668	297.696	-0.009	605.037	605.123	-0.014
3	99.658	103.279	-3.634	202.149	209.926	-3.847
7	43.091	45.878	-6.465	87.061	92.855	-6.655
15	20.713	22.156	-6.966	41.488	45.067	-8.626
31	10.391	11.162	-7.420	20.420	22.458	-9.980
63	5.655	6.217	-9.938	10.747	11.834	-10.114

Table 7.12: Comparison of Predicted and Measured Total Execution Time for Binary Divide-and-Conquer Tasks with Subtasks of Unequal Size

7.4 Comparison of Divide-and-Conquer with Processor Farm

As both divide-and-conquer and processor farm are task-oriented paradigms, it is possible to execute divide-and-conquer applications using the processor farm paradigm. Performance of divide-and-conquer applications executed with *Pfarm* compared to that using *TrEK* depends on the values of the application parameters such as the execution time per task and the total number of tasks. In the following paragraphs, we compare the performance of *TrEK* and *Pfarm* for various values of the parameters. In Table 7.13, we have tabulated the measured total execution time for binary divide-and-conquer applications executed with *TrEK* and *Pfarm* on a binary tree topology.

Divide-and-Conquer strategy performs better compared to processor farm for applications with larger computation time per task. Computation time per task includes all the split and join times in addition to the time required for processing all the subtasks. As shown in Table 7.13, the total execution time taken by *TrEK* is smaller than that taken by *Pfarm* when the computation time per task is greater than 0.766 seconds. This cut-off value depends on the values of the various parameters of the system. This can

	$M = 1000$		$M = 100$	
$T_e(D)$	*TrEK*	*Pfarm*	*TrEK*	*Pfarm*
7.169	117.980	136.315	14.670	21.531
3.072	49.792	58.425	6.475	9.233
1.535	25.175	29.204	3.403	4.620
0.766	13.251	14.585	1.863	2.311
0.190	4.541	3.634	0.632	0.582
0.126	4.769	2.418	0.470	0.390

Table 7.13: Comparison of Total Execution Time for Binary Divide-and-Conquer Applications with *TrEK* and *Pfarm*

be obtained by using the models to calculate and compare the performance of the application using *Pfarm* and *TrEK*. The percentage difference in the total execution time is large when the M is small (see Table 7.13, $M = 100$). Processor farm takes longer for these cases because the wind-down phase is longer in *Pfarm* compared to that of *TrEK*. In *Pfarm*, the wind-down phase begins when there are $4N$ tasks left in the system compared to only $5D - 1$ in *TrEK*, where N is the total number of processors and D is the number of levels of the hardware topology. As the affect of wind-down becomes considerable for applications with fewer tasks, the difference in the total execution time between *Pfarm* and *TrEK* increases.

From the table, it is evident that *Pfarm* works better for applications in which the computation time per task is smaller (less than 0.190 seconds). This is because the overheads involved in splitting and joining in *TrEK* (overheads not present in *Pfarm*) become considerable compared to the execution time per task leading to a larger total execution time compared to that of *Pfarm*. Also, *TrEK* can not be used for applications with small computation time per task (e.g., smaller than 0.126 seconds for a 64-node binary tree topology). This is because the number of levels of the tasks should be greater than the number of levels of the topology, and the base case computation should be greater than the overhead β_f to make use of all the processors effectively.

The only way to use *Pfarm* for executing a single divide-and-conquer problem is for the manager to split the task to obtain a sufficiently large number of independent

subtasks. However, since the manager must do a large number of splits and joins, it can quite easily become the bottleneck. In contrast, since in *TrEK*, the internal nodes do some of the splitting and joining, it is not necessary for the manager to perform as many splits. Note that the manager still must do some splits since otherwise the nodes near the root may idle.

7.5 Chapter Summary

Performance models for divide-and-conquer applications derived in Chapter 6 have been experimentally validated using *TrEK* implementation on a 75-node transputer-based multicomputer. We have described the experiments conducted to determine the values of the system overhead parameters, β_e, β_{f1} and β_{f2}. For a fixed topology, we have experimentally shown that it is better to use a breadth-first spanning tree with a maximum number of leaves. We have validated the models for balanced tree topologies with a large number of experiments varying the values of all the parameters that affect overall performance. As processor farm strategy can be used for some divide-and-conquer applications, we have discussed and compared the performance of using *TrEK* and *Pfarm* for various cases.

Chapter 8

System Integration and Applications

In order to make programming and performance tuning easier, users have to be provided with an integrated environment that includes tools that support all phases of program development and execution, in addition to runtime systems such as *Pfarm* and *TrEK*. In Section 8.1, we briefly describe *Parsec*, an integrated programming environment that provides *Pfarm* and *TrEK* with supporting tools such as a graphical interface, mapper, loader and debugger on our transputer-based system. We discuss how this integrated environment supports reusability, reconfigurability and performance tuning. In section 8.2, we discuss the techniques that can be used to obtain the values of application dependent parameters in order to use the models for performance tuning. Finally, we describe two applications that have been developed using *Pfarm* and *TrEK*.

8.1 *Parsec*: An Integrated Programming Environment

Pfarm and *TrEK* provide programming templates to efficiently execute applications that fit into processor farm and divide-and-conquer paradigms, respectively. In order to make it easier for application programmers to use these templates on a multicomputer system, it is important to provide a programming environment that supports all phases of program development and execution. Such a programming environment should not only have a variety of tools that help programmers in developing, executing, debugging

and tuning a parallel program, but must support their cooperative functioning through close integration. The following facilities should be present in an integrated programming environment to effectively support *Pfarm* and *TrEK* on a multicomputer:

- an interface that hides both the system hardware and software complexities,
- support for reusability,
- support for easy reconfigurability of the system,
- support for loading and executing of programs,
- performance monitoring and tuning facilities based on the performance models, and
- program debugging tools.

Initial work in developing an integrated programming environment that addresses the above requirements on a large transputer-based system has been reported in [CGJ+91, FSWC92]. *Parsec* is an on-going project at UBC to support creating templates or applications, and includes tools for building, mapping and loading the program onto the system. *Pfarm* and *TrEK* have influenced the design of *Parsec.* In order to make performance tuning easier for applications using *Pfarm* or *TrEK*, *Parsec* supports parameterized process graphs. A parameterized process graph is a family of interconnection networks with one or more parameters that control structural properties. In addition, *Parsec* allows users to change these parameters in an easier way. Thus, a user can easily run an application on its optimal N and topology, once they are determined from the models.

In *Parsec*, a "template implementor" describes a template in terms of a parameterized process structure which is then turned into a system module. Users of a template do not have to understand the details of its implementation. They simply instantiate a copy of the template and provide any necessary parameters and code. *Parsec* creates all the files (makefiles, configuration files, and load scripts) necessary for running the application.

Pfarm and *TrEK* templates have been incorporated into *Parsec* and they make use of the parameterized graph structure to simplify scaling and restructuring of the system. Within this programming environment, currently only Trollius is available to the programmers. The following discussion focuses on how the various tools in the environment support programming of applications that use the *Pfarm* template under *Parsec*.

In *Parsec*, programmers are provided with an easy-to-use graphical interface to *Pfarm* and *TrEK*, and to the system in general. The interface, developed by Feldcamp [FW93] is an X windows application utilizing the OpenLook GUI. Figure 8.1 shows the graphical interface provided to application programmers when *Pfarm* template is selected. Programmers can easily modify the template by including the files that contain the application dependent code. *Parsec* supports system reconfigurability in an easier way through the graphical interface. The user can change the parameters (such as degree and depth) that define the topology to be used for executing an application with *Pfarm*. Then, the programmer can build the object files needed to execute the application by using the makefiles generated by *Parsec*. These makefiles remove the concerns of choosing the right compiler and libraries from the user. Users can easily include any additional libraries, if necessary. To execute a *Pfarm* application, two different object codes have to be built, one for the manager and another for all the workers.

After choosing the topology to be used, the user must map this topology onto the 75-node transputer based hardware system. The mapping tool [Mul93] inputs a description of the hardware, processors and crossbars, and outputs a crossbar setting, a process to processor assignment, and a configuration file. The mapping tool uses a greedy algorithm to do the mapping. In the case of *Pfarm* and *TrEK*, a different notion of mapping is needed. *Pfarm* and *TrEK* need one-to-one mapping of workers on to the processors, without any dilation. Also, on a fixed interconnection network, the mapper should be able to map the workers onto a breadth-first spanning tree.

Parsec includes a loader tool [Mul93] that builds the network configuration file based on the mapping obtained by the mapping tool. In addition, the loader generates a script that is used to execute the application program. This script includes the Trollius commands to boot the network and to load the appropriate programs onto the transputer

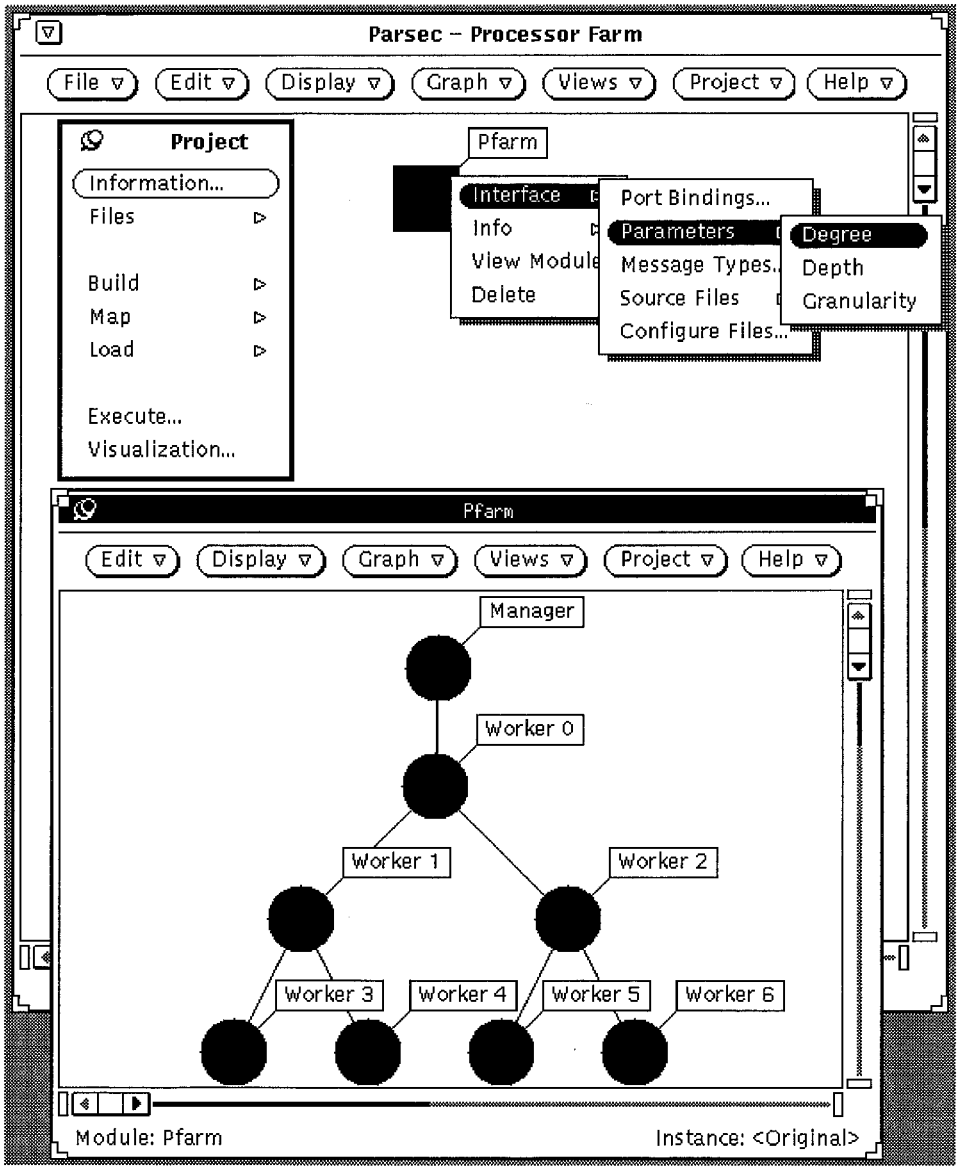


Figure 8.1: Graphical Interface to *Pfarm* in Parsec

nodes. Users execute the application program by running this load script. The loader allows users to choose either the network or physical level of communication. The network level is slower compared to the physical level, but unlike the physical level, users are able to print from any node and to monitor the state of the processes on each node. This allows users to choose network level during the program development and debugging phases, and then use physical level to obtain better performance.

8.2 Performance Tuning

In this section, we describe how the programmer can use models for performance prediction and tuning.

8.2.1 Parameter Measurements

In order to use the models for performance prediction and tuning, one has to determine the input parameters to the model. The user must supply the values for all the parameters other than the system overhead parameters (β_e and β_f in case of *Pfarm*, and β_e, β_{f1} and β_{f2} in case of *TrEK*). The values of these overhead parameters are obtained once using the techniques described in Sections 5.1 and 7.1 for *Pfarm* and *TrEK* respectively. Here, we explain the techniques that are useful in determining the values of the application dependent parameters.

Processor Farm

In the processor farm case, the application dependent parameters that affect the overall performance are: the average execution time per task(T_e), the total number of tasks (M), the data size (d) and result size (r) per task. An application programmer generally knows the values of M, d and r for an implementation, otherwise these values can be easily obtained. Obtaining the value of T_e is not so straightforward as it depends on the nature of the application program in addition to the implementation. Here, we briefly describe the different techniques that can be used to obtain T_e.

1. In the case of application programs that consist of tasks, each of which require the same amount of computation, the easiest way to measure T_e is to scale down the program to a single task or a small number and execute it with *Pfarm* on a single worker node. Then, T_e can be obtained from the expression $T_{total} = M(T_e + \beta_e)$ using the measured total execution time and the value of β_e.

   This technique can also be used for application programs that consist of tasks with varied computation requirements. In this case, one can determine average T_e by finding T_e for a representative set of tasks. If it is difficult to choose right set of tasks, then, the third technique can be used.

   If the application program consists of multiple phases, where all the tasks in a phase belong to the same type, then as explained in the robustness section 5.4, it is necessary to calculate a separate T_e for each phase.

2. If there is a direct relationship between the input data size of a task and its computation requirement, then, by determining T_e for a certain data size, one can estimate T_e for a new data size based on the relation.

3. If T_e cannot be obtained by either of the previous techniques, then it can be determined by executing *Pfarm* on a smaller number of processors and using the model (with known N and T) to calculate T_e. If the performance obtained is not equal to that of the communication bounds, then, models can be used to obtain the average value of T_e by plugging in the total number of tasks and the measured total execution time.

Divide-and-Conquer

In the case of *TrEK*, the application dependent parameters that affect the performance are: the execution, split and join times ($T_e(i), T_s(i), \text{and} T_j(i)$) at each level of the divide-and-conquer task, the total number of tasks (M), the data size (d), and the result size (r) per task. As in the *Pfarm case*, an application programmer generally knows the values of M, d and r for an implementation. Obtaining the values of the execution, split and join times is not so straightforward.

As explained in Section 6.2, the computational requirement of a fixed degree divide-and-conquer task (or subtask) with an input data size of n can be expressed as

$$W(n) = split(n) + join(n) + kW(n/k), \tag{8.1}$$

where $split(n)$ is the splitting cost for a task with size n, $join(n)$ is the joining cost to produce a result of size n and k is the degree of the divide-and-conquer tasks to be processed. $T_e(i), T_s(i)$ and $T_j(i)$ are given by $W(n), split(n)$ and $join(n)$, respectively, depending on the data size (n) of the tasks at the ith level. Thus, in order to use the performance models, one has to determine $W(n), split(n)$ and $join(n)$ for the given application program.

In general, $W(n)$ can be expressed by the time complexity of the algorithm such as $\mathcal{O}(n \log n)$ or $\mathcal{O}(n^2 \log n)$. Thus, we have to find the constants underlying these time complexities to get the value of $W(n)$, the time needed to execute the task on a processor. This can be determined by executing a scaled down version of the problem as in the first technique outlined for measuring T_e in the processor farm case. This experiment can be repeated with few different input sizes to verify the values of the constants. Similar measurements can be used to obtain the values for the constants to be used for split and join times.

8.2.2 Performance Analysis Library

To make it easier to use the models for prediction and tuning, application programmers are provided with a set of performance analysis library functions for both *Pfarm* and *TrEK*. These functions accept the values of application dependent parameters and output the predicted performance metrics such as throughput, speedup and total execution time.

8.3 User Interface

Pfarm and *TrEK* were designed to hide the underlying complexities of the multicomputer system from the user. All the system dependent code is in the execution kernels and the user has to concentrate only on the application dependent code. The kernel can be used for application programs that fit the corresponding parallel programming paradigms.

Both *Pfarm* and *TrEK* are run time kernels where the user code is linked with the system code to produce a single executable object for each processor node. In the manager, the user invokes the system routines to submit tasks and receive back results. In the workers, program control lies within the system code and the system invokes the user code at the appropriate times.

8.3.1 *Pfarm*

In the case of *Pfarm*, there are two different executables, one for the manager node and the other for the worker nodes. The user part of the manager code consists of the following functions:

1. `master_init()` - The system code calls this function at the beginning of the execution. This function consists of the initialization part of the user code, such as reading data from an input file, etc.. Also, if there is any global data to be broadcast to all the worker nodes, the user code can initiate the system call, `bc_send()` to do the broadcast.

2. `data_generator()` - This function consists of the part of the user code that generates the tasks. In real-time applications, it could receive data from some device and generate the corresponding tasks. The tasks are passed to *Pfarm* for processing by the system call `do_task()`.

3. `result_receiver()` - This function consists of the part of the user code that collects the results. The system call `get_result()` returns the next available result. This function could also include any processing of the results.

Both `data_generator()` and `result_receiver()` are called by low priority system processes. These functions are called in the beginning of the execution after creating all the processes and are run concurrently until they finish their respective jobs. In the case of applications in which later tasks depend on the results of the initial tasks, the user program could consist of only one function, `data_generator()`. This function performs both the system calls, `do_task()` and `get_result()`.

The user part of the worker code consists of the following two functions:

- `slave_init()` - This function consists of any initialization part of the user code needed on each worker node. Also, if there is any broadcast of the global data, this function can do the system call, `bc_receive()` to receive the broadcast data.
- `comp_fn()` - This function contains the user code to process a task. It is called by the worker process and takes a pointer to the task data and returns a pointer to the result and the size of the result.

Pfarm provides the following system calls:

1. `do_task(task_type, task_size, task_ptr)`
2. `result_ptr get_result()`
3. `bc_send(bc_data_size, bc_data_ptr)`
4. `bc_data_ptr bc_receive()`

8.3.2 *TrEK*

TrEK provides the same system calls to the user as in the *Pfarm* case. The user part of the manager code to be run in the *TrEK* case is similar to that in the *Pfarm* case. The user part of the worker code includes the following two functions in addition to the `comp_fn()` described in the *Pfarm* case.

- `split_fn()` - This function consists of the user code that splits a task and is called by the split process in the *TrEK*. It takes a pointer to the task as input and returns the pointers to the subtasks.
- `join_fn()` - This function consists of the user code that combines the results of subtasks and is called by the join process in the *TrEK*. It takes the pointers to the subresults as input and returns a pointer to the combined result.

As an example, of the user code for an FFT implementation that uses *TrEK* has been included in the following section.

8.4 Applications

Several applications have been developed using *Pfarm* and *TrEK* on our transputer-based system by other graduate students and myself. In this section, we discuss two interesting cases where the models were used to understand the performance of applications. Results reported here are for Logical systems versions.

8.4.1 Cepstral filtering

Pfarm was used to parallelize a vision application that performs Cepstral filtering for motion analysis [BL93]. It takes two images of the same subject at different instances of time and determines the motion of the subject. The user code on the manager consists of two parts, a data-generator and a result-receiver. The data generator function partitions the images into small blocks and puts two corresponding blocks, one from each of the two images, into a single task. The result-receiver collects the results of the partial motion analysis and assembles them. In the following paragraph, we discuss how the models were used for performance tuning.

Initially, the program was tested using two smaller images of 64×64 bits. The task execution time (T_e) for images divided into blocks of 16 bits was measured by executing the program on a single worker node. For this case, the value of T_e was 178.048 ms. We were interested in using the program with larger images of 512×512 bits. The performance model was used to determine the best topology and the number of nodes to run this application for larger images. The model predicted that a 63-node balanced binary tree gives maximum performance with the total execution time of 3.141 seconds. We executed the application program on a 63 node binary tree and found that it took 4.492 seconds, which was considerably larger than the predicted value. To investigate the reasons for the large error, we ran the program on a linear chain and a ternary tree. The model predicted that on a 64 node linear chain, it would take 4.743 seconds and on a 40 node ternary tree 4.813 seconds. For these cases, the prediction was accurate since the measured total execution times were 4.517 and 4.756 seconds for chain and ternary tree cases respectively. Then, the program was executed on a 32 node binary tree and

the prediction was found to be accurate.

To investigate the reasons for the large error in the prediction for the 63-node binary tree case, we recorded the number of tasks executed on each node. We found that the leaf nodes executed a fewer tasks than the intermediate nodes. This occurs only when the system is communication bound, but according to the models, the system should not have reached the communication bound (based on the data and result sizes of the tasks). The only explanation for this scenario is that the system violated one of the assumptions in the model. On examination, we hypothesized that the data generator may not be generating the tasks at the rate at which the system can receive and process them. Therefore, we ran several experiments to measure the rate at which the data generator was generating the tasks, and found that the rate was 248.32 tasks/sec. However, according to the model, a 63-node binary tree for this application program can process tasks at the rate of 349.06 tasks/sec. This confirmed our suspicion that the large error occurred because the data generator was unable to keep up a continuous flow of tasks into the farm. For chain, ternary tree and smaller binary tree topologies the predictions were accurate because the task processing rate of these topologies were smaller than the rate at which tasks were generated.

8.4.2 Fast Fourier Transform (FFT)

Divide-and-conquer strategy has been used to design efficient sequential FFT algorithms. In this example, we have used *TrEK* to parallelize a sequential algorithm that uses FFT for multipoint evaluation of a polynomial over a field [Sed83]. For this implementation, execution time starts increasing for binary trees with more than 4 levels as the throughput reaches the communication bound given by equation (6.18) between levels 3 and 4.

The sequential FFT algorithm [Sed83] is reproduced below:

```
Algorithm FFT(N, a(x), w, A)
if N = 1 then
    A_0 := a_0;
else
    /* split */
    n := N/2;
```

$b(x) := \sum_{i=0}^{n-1} a_{2i}x^i$;
$c(x) := \sum_{i=0}^{n-1} a_{2i+1}x^i$;
/* recursive calls */
FFT$(n, b(x), w^2, B)$;
FFT$(n, c(x), w^2, C)$;
/* combine */
for $k := 0$ **to** $n-1$ do
$A_k := B_k + w^k C_k$;
$A_{k+n} := B_k - w^k C_k$;
endfor
endif

In order to show the interaction between *TrEK* and the user code, the user code that implements this FFT algorithm has been included.

```
data_generator()
{
for (i=1; i<=TotalJobs; i++) {
/* prepare the data for a task */
/* send the task to TrEK by calling do_task */
   do_task(task_type, user_data_size, ptr);
}}

result_receiver()
{
user_result *ptr;
for (i=1; i<=TotalJobs; i++) {
/* get the next available result */
   ptr = (user_result *) get_result();
/* process the result */
}}

comp_fn(user_data *ptr, int *ur_size, char **ur_ptr)
{
/* call the fft function */
fft1(N, fptr, A);
/* set the argument values to be returned*/
*ur_size = user_result_size;
*ur_ptr = (char*) p;
}
```

```
split_fn(ptr, split_datasize, ptrs)
user_data *ptr;
int *split_datasize;
char *ptrs[];
{
/* split the task */
/* set the argument values to be returned*/
ptrs[0] = (char *) ptr1;
ptrs[1] = (char *) ptr2;
*split_datasize = user_data_size;
}

join_fn(jb_userres, jres_size, ptr)
char *jb_userres[TASK_DEG];
int *jres_size;
char **ptr;
{
/* join the results */
/* set the argument values to be returned*/
*ptr = (char *) p;
*jres_size = user_result_size;
}
```

After parallelizing this algorithm using *TrEK*, we ran the program on a single node with smaller N (32 and 64), and used the technique described in Section 8.2 to determine the values of the application dependent parameters. As the sequential algorithm is an $\mathcal{O}(NlogN)$ algorithm, we set

$$T_e(N) = aN + bNlogN.$$

Because the split and join costs in this algorithm are $\mathcal{O}(N)$, we set

$$T_s(N) + T_j(N) = c + dN.$$

We calculated the values of the constants (a, b, c and d) for this implementation using the measured execution times for smaller N. The values of these constants are: $a = 0.000330$, $b = 0.000091, c = 0.001695$ and $d = 0.000082$.

The models were used to predict the performance of this implementation for larger N (128, 256, 512 and 1024). From the models, we found that the throughput for any N

Problem Size N	Lower bound Time	Measured Time	Measured Speedup
128	13.55	16.841	7.32
256	24.05	28.620	9.46
512	45.06	52.864	11.13
102	87.06	102.820	12.42

Table 8.1: Experimental results for FFT on a 16-node binary tree

reaches the communication bound given by equation (6.18) for binary trees of 4 levels and ternary trees of 3 levels. The system reaches this bound because of the split and join costs at the root processor. If a larger tree is used, the root processor would be unable to split and forward the tasks at the rate in which the rest of the processors can process the tasks. The only way to improve the performance in this case is to optimize the code for split and join functions.

We verified these predictions by experimenting with larger N (see Table 8.1). As predicted, the measured total execution time did not decrease when we increased the size of a binary topology from 4 levels to 5 levels. Actually, the measured execution time increased because of the reason described in Section 7.3.4.

Chapter 9

Conclusions

This dissertation has explored a parallel programming approach that addresses the need of providing a programming environment that is easy to use, efficient and supports performance tuning on multicomputers. In this approach, users are provided with programming support based on parallel programming paradigms. We have studied two commonly used parallel programming paradigms: processor farm and divide-and-conquer. Runtime system support for these two paradigms are designed such that they are easy-to-use and can maximize the performance for applications that fit these paradigms. Performance models are derived for these systems taking into account the computation and communication characteristics of the applications that fit the paradigm in addition to the characteristics of the hardware and software system. The models determine the parameters that affect the performance and can be used for performance prediction and tuning. This work has contributed to our understanding of these systems and their limitations.

In designing reusable and efficient runtime systems, many trade-offs have to be considered. In Chapter 4 and 6, we have described the trade-offs involved in the design of *Pfarm* and *TrEK* respectively. Hiding the complexities of the underlying hardware and software system is the major consideration in the design of these runtime systems. These systems include all the necessary code for the system dependent issues such as communication, synchronization, task scheduling, and load balancing. Thus, users can concentrate on the application dependent compute intensive code. In order to be efficient, runtime systems are designed to make use of all the available parallelism in the

hardware system such as the ability to simultaneously communicate on all the links. The system overheads that limit the overall performance of the applications are kept to a minimum. Both systems implement distributed dynamic task scheduling strategies so that they can work well even for applications that can be decomposed into tasks with varying computational requirements. The systems are designed such that they are topology independent, i.e., they can scale and run on any processor topology.

It is difficult to obtain a single performance model that can be used for all the applications on a parallel system. However, it is possible to derive good performance models for each of the virtual machines as every paradigm is a restricted model of parallel computation. Performance models for processor farm and divide-and-conquer virtual machines have been derived in Chapter 4 and 6 respectively. These models take into account the computation and communication characteristics of the applications that fit the paradigm in addition to the characteristics of the hardware and software system. General analytical frameworks that can be used to predict the performance on any tree topology have been presented for both of these paradigms. As both of these task-oriented systems behave like a pipeline, it is important to analyze start-up and wind-down.

For the processor farm case, we have shown that, on a fixed topology, a breadth-first spanning tree provides maximum performance and steady-state performance of all breadth-first spanning trees are equal. As balanced tree topologies provide maximum performance in the case of reconfigurable systems, we have derived performance models for these topologies using the general analytical framework.

TrEK can execute divide-and-conquer computations of any degree and depth on any arbitrary tree topology. Unlike idealized parallel implementations of divide-and-conquer algorithms on tree processors [HZ83, Col89], *TrEK* allows intermediate processors to do subtask processing to make use of all the available parallelism in the hardware system. The analytical framework assumes a flow of divide-and-conquer tasks. As explained in Section 3.4, this framework works well even for applications that consist of a single divide-and-conquer computation with large degree and depth compared to the underlying hardware topology. Experimentally, we have found that, on a fixed topology, a breadth-first spanning tree with maximum number of leaves obtains maximum performance. As

balanced tree topologies provide maximum performance in the case of reconfigurable systems, we have derived models that can predict performance of any fixed k-ary divide-and-computations on any g-ary balanced tree topology.

Pfarm and *TrEK* have been implemented on a 75 node transputer-based multicomputer. They are implemented using C on two different software environments: Logical Systems and Trollius. As *Pfarm* and *TrEK* provide standard interfaces to the user code irrespective of the environment they are implemented on, the user code is portable from one system to the other. Performance models have been experimentally validated using *Pfarm* and *TrEK*. The models are found to be accurate as reported in Chapters 5 and 7. In order to use the models, it should be possible to determine the values of the parameters in an easy way. We have explained the techniques that can be used to determine the values of the system dependent and application dependent parameters. We have discussed how the models can be used in predicting and tuning the performance.

It is possible to use the processor farm strategy to parallelize some divide-and-conquer applications. Divide-and-conquer strategy performs well compared to the processor farm strategy for applications with larger tasks and for those that consist of a smaller number of tasks. Performance models can be used to determine the strategy to be used for a given application.

In order to make it easier for application programmers to use runtime systems on a multicomputer, they must be provided with a programming environment that supports all phases of program development and execution. In Chapter 8, we have described such an integrated programming environment, *Parsec*, developed on our transputer-based multicomputer. In addition to providing *Pfarm* and *TrEK* templates to application programmers, *Parsec* supports tools such as a graphical interface, mapper, loader and debugger We have discussed how *Parsec* supports reusability, reconfigurability and performance tuning.

9.1 Future Directions

This research can be continued in several directions to further explore the usefulness of this approach for parallel programming.

We have mentioned that the same design can be used for implementing *Pfarm* and *TrEK* on any distributed memory parallel computer that has the characteristics detailed in Chapter 3. Also, the performance models derived here can be used for any system that satisfies the assumptions used in the models. By implementing *Pfarm* and *TrEK* on two different software environments, Logical Systems and Trollius, we have shown that the user programs are portable and the same models can be used for both implementations using appropriate parameter values. The claims of being able to use the same design and modeling on any multicomputer system in addition to user programs being portable can be further strengthened by implementing the runtime systems on other hardware platforms, such as C40 based machines. Developing more application programs with these runtime systems in several application areas could further support the usefulness of these runtime systems.

It is interesting to research the expressiveness of the task-oriented paradigms, processor farm and divide-and-conquer, i.e., whether these paradigms and the associated implementations can be modified or enhanced to make use of them for applications that may not exactly fit the underlying computational models.

Pfarm system can be used for applications with differing characteristics as discussed in Chapter 8. It is possible to modify the same design to develop a runtime system that can be used for the Task Queue paradigm. The applications that fit the Task Queue paradigm consist of an initial set of tasks, which could be allocated to various processors in the system. These tasks may generate new tasks that have to be processed. Unlike in the case of divide-and-conquer, the results of these new tasks need not be joined by the parent. In this case, the task scheduling and load balancing has to be different from that of *Pfarm*. Each processor can keep a local task queue to which all the new tasks are added. It can exchange the load information with its neighbors and transfer the tasks to the lightly loaded neighbors, if necessary. Deriving performance models for such systems

may need probabilistic models that reflect the task generation.

It is possible to expand this approach to develop virtual machines that support applications that fit other parallel programming paradigms such as Compute-Aggregate-Broadcast, Systolic and Dynamic Programming. Also, in practice, some applications may consist of several phases each of which may need different programming paradigms. This work can be expanded by developing programming environment that supports such applications. There are many issues to be considered here such as how the different systems exchange the data and results, whether they can be co-existent on the system or they should be interleaved.

Bibliography

[ACG91] S. Ahmed, N. Carriero, and D. Gelernter. The Linda Program Builder. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing, Research Monographs in Parallel and Distributed Computing.* MIT Press, Cambridge, Massachusetts, 1991.

[AG89] D. Atapattu and D. Gannon. Building Analytical Models into an Interactive Performance Prediction Tool. *Proc. Supercomputing '89*, pages 521–530, November 1989. Indiana U.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[Alv90] G. A. Alverson. Abstractions for Effectively Portable Shared Memory Parallel Programs. Technical Report 90-10-09, Dept. of Computer Science and Engineering, University of Washington, October 1990.

[Amd67] G. M. Amdahl. Valildity of the Single-processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings.* AFIPS Press, Reston, Va., 1967.

[Arl88] R. Arlauskas. iPSC/2 System: A Second Generation Hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 38–50. ACM Press, January 1988.

[AS88] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, August 1988.

[BBFB89] Moshe Braner, Gregory D. Burns, David L. Fielding, and James R. Beers. Trollius OS for Transputers. In D. Stiles, editor, *Transputer Research and Applications (NATUG 1)*, 1989.

[Ben80] J.L. Bentley. Multidimensional Divide-and-Conquer. *Commun. ACM*, 23:214–229, 1980.

[BFKK91] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. In *Proc. 3th ACM Symp. on Principles and Practice of Parallel Programming (PPOPP)*, pages 213–223, April 1991.

[BL93] E. Bandari and J. J. Little. Multi-evidential Correlation & Visual Echo Analysis. Technical Report TR 93-1, Department of Computer Science, University of British Columbia, Vancouver, Canada, January 93.

[BTU88] R. D. Beton, S. P. Turner, and C. Upstill. A State-of-the-Art Radar Pulse Deinterleaver - A Commercial Application of Occam and the Transputer. In C. Askew, editor, *Occam and the Transputer - Research and Applications*, pages 145–152. IOS Press, 1988.

[CCT93] A.G. Chalmers, N.W. Campbell, and B.T. Thomas. Computational Models for Real-Time Tracking of Aircraft Engine Components. In S. Atkins and A. Wagner, editors, *Transputer Research and Applications 6 (NATUG 6).* IOS Press, May 1993.

[CG89] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, pages 323–357, September 1989.

[CGJ+91] S. Chanson, N. Goldstein, J. Jiang, H. Larsen, H. Sreekantaswamy, and A. Wagner. TIPS: Transputer-based Interactive Parallelizing System. In *Transputing '91 Conference Proceedings, Sunnyvale, Calf.* IOS Press, April 1991.

[CHvdV88] N. Carmichael, D. Hewson, and J. van der Vorst. A Prototype Simulator Output Movie System based on Parallel Processing Technology. In C. Askew, editor, *Occam and the Transputer - Research and Applications*, pages 169–175. IOS Press, 1988.

[CK88] D. Callahan and K. Kennedy. Compiling Programs for Distributed-memory Multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[CK91] K. M. Chandy and C. Kesselman. Parallel programming in 2001. *IEEE Software*, pages 11–20, November 1991.

[Cok91] Ronald S. Cok. *Parallel Programs for the Transputer.* Prentice-Hall, 1991.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, Massachusetts, 1989.

[CU90] I. Cramb and C. Upstill. Using Transputers to Simulate Optoelectronic Computers. In S.J. Turner, editor, *Tools and Techniques for Transputer Applications (OUG 12).* IOS Press, April 1990.

[Dav85] R. E. Davis. Logic Programming and Prolog. *IEEE Software*, pages 53–62, September 1985.

[Den80] J. B. Dennis. Data Flow Supercomputers. *Computer*, pages 48–56, November 1980.

[Dun90] R. Duncan. A Survey of Parallel Computer Architectures. *IEEE Computer*, pages 5–16, February 1990.

[EZL89] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, March 1989.

[F+90] D.L. Fielding et al. The Trollius Programming Environment for Multicomputers. In A. Wagner, editor, *Transputer Research and Applications (NATUG 3)*. IOS Press, April 1990.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors (vol. 1)*. Prentice Hall, New Jersey, 1988.

[FK89] H. P. Flatt and K. Kennedy. Performance of parallel processors. *Parallel Computing*, 12:1–20, 1989.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, C-21(9):948–960, September 1972.

[FO90] I. Foster and R. Overbeek. Bilingual Parallel Programming. In A. Nicolau, D. Glelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compiler for Parallel Processing, Research Monographs in Parallel and Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1990.

[FSWC92] D. Feldcamp, H. V. Sreekantaswamy, A. Wagner, and S. Chanson. Towards a Skeleton-based Parallel Programming Environment. In A. Veronis, editor, *Transputer Research and Applications NATUG 5*. IOS Press, April 1992.

[FT90] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice Hall, New Jersey, 1990.

[FW93] D. Feldcamp and A. Wagner. Parsec: A Software Development Environment for Performance Oriented Parallel Programming. In S. Atkins and A. Wagner, editors, *Transputer Research and Applications 6 (NATUG 6)*. IOS Press, May 1993.

[Gab90] E. Gabber. VMPP: A Practical Tool for the Development of Portable and Efficient Programs for Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):304–317, July 1990.

[Gal90] J. Galletly. *Occam2*. Pitman Publishing, 1990.

[GC92] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Commun. ACM*, 35(2):97–107, February 1992.

[GK92] A. Gupta and V. Kumar. Analyzing Performance of Large Scale Parallel Systems. Technical Report TR 92-32, Dept. of Computer Science, University of Minnesota, October 1992.

[GKP89] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, Mass., 1989.

[GM85] D. H. Grit and J. R. McGraw. Programming Divide and Conquer for a MIMD Machine. *Software-Practice and Experience*, 15(1):41–53, January 1985.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, 1988.

[Gus88] J. L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, May 1988.

[Hey90] A. J. G. Hey. Experiments in MIMD Parallelism. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE: Parallel Architectures and Languages Europe*, pages 28–42. Springer-Verlag, New York, 1990. Lecture Notes in Computer Science 366.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler Support for Machine-independent Parallel Programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors.* 1991.

[Hom93] D. Homeister. An Adaptive Granularity Scheduler for Multiprocessors. In S. Atkins and A. Wagner, editors, *Transputer Research and Applications 6 (NATUG 6).* IOS Press, May 1993.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[HZ83] E. Horowitz and A. Zorat. Divide-and-Conquer for Parallel Processing. *IEEE Transactions on Computers*, 32(6):582–585, June 1983.

[Inc91] Texas Instruments Incorporated. *TMS 320C4X User's Guide.* 1991.

[Inc92] Texas Instruments Incorporated. *TMS 320C4X Parallel Runtime Support Library: User's Guide.* 1992.

[K+87] H.T. Kung et al. The Warp Computer: Architecture, Implementation, and Performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, December 1987.

[K+88] H.T. Kung et al. iWARP: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88, Orlando, FL.* IEEE, Computer Society Press, 1988.

[K+90] H.T. Kung et al. Supporting systolic and memory communication in iWarp. In *Proceedings of the 17th Intl. Symposium on Computer Architecture.* IEEE, Computer Society Press, 1990.

[Kar87] A. H. Karp. Programming for Parallelism. *IEEE Computer*, pages 43–57, May 1987.

[KF90] A. H. Karp and H.P. Flatt. Measuring Parallel Processor Performance. *Commun. ACM*, 33(5):539–543, May 1990.

[Kog85] P. M. Kogge. Function-based Computing and Parallelism: A Review. *Parallel Computing*, 2:243–253, 1985.

[KR87] V. Kumar and V. N. Rao. Parallel Depth-first Search, Part II: Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[KT88] M. Kallstorm and S. S. Thakkar. Programming thre Parallel Computers. *IEEE Software*, 5(1):11–22, Jan 1988.

[Kun89] H. T. Kung. Computational Models of Parallel Computers. In R. J. Elliot and C.A.R. Hoare, editors, *Scientific applications of multiprocessors.* Prentice Hall, 1989.

[Lim84] INMOS Limited. *Occam Programming Manual.* Prentice Hall International, New Jersey, 1984.

[lim87] Meiko limited. *Computing Surface Technical Specifications.* Meiko Ltd., Bristol, UK, 1987.

[Lim88] INMOS Limited. *Transputer Development System.* Prentice Hall, New Jersey, 1988.

[Moc88] Jeffrey Mock. Process, channels and semaphores (version 2). Logical Systems C Programmers Manual, Logical Systems, 1988.

[MS88] D. L. McBurney and M. R. Sleep. Transputer-based Experiments with the ZAPP Architecture. In J. W. de Bakker et al, editor, *Lecture notes in Computer Science.* 1988.

[Mul93] S. Mulye. Communication Optimization in Parsec: A Programming Environment for Multicomputers. Master's thesis, Dept. of Computer Science, University of British Columbia, 1993. in preparation.

[Nel87] P. A. Nelson. *Parallel Programming Paradigms.* Ph.D. thesis, Department of Computer Science , University of Washington, Seattle, WA, 1987.

[NRFF93] D.F. Garcia Nocetti, M.G. Ruano, P.J. Fish, and P.J. Fleming. Parallel Implementation of a Parametric Spectral Estimator for a Real-time Doppler Flow Detector. In S. Atkins and A. Wagner, editors, *Transputer Research and Applications 6 (NATUG 6).* IOS Press, May 1993.

[Pri87] D. Pritchard. Mathematical Models of Distributed Computation. In *Proceedings of the 7th Occam User Group Technical Meeting.* IOS, 1987.

[Pri90] D. J. Pritchard. Performance Analysis and Measurement on Transputer Arrays. In Aad van der Steen, editor, *Evaluating Supercomputers.* Chapman and Hall, 1990.

[PW86] D. A. Padua and M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12):1184–1201, December 1986.

[RSV90] M. Rao, Z. Segall, and D. Vrsalovic. Implementation Machine Paradigm for Parallel Programming. In *Proceedings of Supercomputing '90*, pages 594–603. IEEE, Computer Society Press, 1990.

[SCW92] H. V. Sreekantaswamy, S. Chanson, and A. Wagner. Performance Prediction Modeling of Multicomputers. In *Proceedings of the Twelth International Conference on Distributed Computing Systems (ICDCS)*, June 1992.

[Sed83] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, 1983.

[Sei85] C.L. Seitz. The cosmic cube. *Commun. ACM*, 28(1):22–33, Jan 1985.

[Son93] Y. Song. The Implementation of Sequential and Parallel Algorithms for Solving Almost Block Bidiagonal Systems. Master's thesis, Dept. of Computer Science, University of British Columbia, March 1993.

[SR85] Z. Segall and L. Ruldolph. PIE: A Programming and Instrumentation Environment for Parallel Processing. *IEEE Software*, 2(6):22–37, November 1985.

[SS91] S. S. Sturrock and I. Salmon. Application of Occam to Biological Sequence Comparisons. In J. Edwards, editor, *Occam and the Transputer - Current Developments*, pages 181–190. IOS Press, 1991.

[Sto87] Q. F. Stout. Supporting Divide-and-Conquer Algorithms for Image Processing. *Journal of Parallel and Distributed Computing*, 4:95–115, February 1987.

[Sto88] H. Stone. *High Performance Computers.* Addison-Wesley, 1988.

[Sys90] Logical Systems. C Programmers Manual. Corvallis, OR, 1990.

[TD90] R.W.S. Tregidgo and A.C. Downton. Processor Farm Analysis and Simulation for Embedded Parallel Processing Systems. In S.J. Turner, editor, *Tools and Techniques for Transputer Applications (OUG 12).* IOS Press, April 1990.

[Ull84] J.D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, MD, 1984.

[WSC93] A. Wagner, H. Sreekantaswamy, and S. Chanson. Performance Issues in the Design of a Processor Farm Template. In *Proceedings of the World Transputer Conference.* IOS Press, 1993. To appear.

Glossary

β_e	Average processor overhead for a locally processed task.
β_f	Average processor overhead for a forwarded task.
τ	Communication rate of the links.
f_i	The fraction of the total number of tasks executed by a processor i.
M	Total number of tasks in an application program.
N	The total number of processors in the system.
S_D	Throughput of a D-level tree.
SP_D	Speedup of a D-level tree.
T_{cd}	Average communication time required to transfer a task from a processor to its neighbor.
T_{cd}	Average communication time required to transfer a result from a processor to its neighbor.
T_e	Average processing time per task in the processor farm case.
$T_e(i)$	Average processing time per task or subtask at the ith level of the hardware topology in the divide-and-conquer case.
$T_j(i)$	Average result joining time at the ith level of the hardware topology in the divide-and-conquer case.
$T_s(i)$	Average task splitting time at the ith level of the hardware topology in the divide-and-conquer case.
T_{ss}	Steady-state execution time.
T_{su}	Start-up time.
T_{total}	Total execution time.
T_{wd}	Wind-down time.
V_i	Number of tasks or subtasks that visit processor i.