SCHEDULING DATA TRANSFERS IN PARALLEL COMPUTERS AND COMMUNICATIONS SYSTEMS

by

RAVI KUMAR JAIN, B. Sc., M. S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1992

To my parents and my sister

S. K. Jain, S. Jain, and Poonam

Acknowledgments

It is politic to begin a dissertation acknowledgement with a formal statement of thanks to the advisor. However, my cynicism about academic conventions is completely overshadowed by the fact that I have truly been fortunate to have had not one, but two, of the best advisors a graduate student could hope for: Jim Browne and John Werth. I am grateful for their support and guidance, of course; but more importantly I appreciate their confidence in me and, dare I say it, their encouraging me to believe in myself. I especially appreciate John Werth, not only for his technical insight and the countless long hours he has spent with me, but for being the warm, wise and wry human being that he is.

I would also like to thank Galen Sasaki for his technical guidance, particularly during the early stages of this work. Jeff Brumfield, while not on my committee, has been a great source of advice and support. Thanks are due to Al Mok and Allen Emerson for serving on my committee.

There have been many friends and colleagues who have helped me survive the graduate school game at Texas. I owe thanks to Michael Barnett, Ted Briggs, Rick Froom, Peter Newton, J. R. Rao, and Ravi Rao, among others. Life in Austin was made possible by Dilip D'Souza, Carla Feldpausch, Julia

Fitzgerald, Veena Gondhalekar, Pradeep Jain, Regina Lauderdale, Claire Loe, Ricardo Salvatore, and others; their friendship nourished me through many academic winters, and made the rat race bearable. I owe a great deal to the caring and support of my family in Austin: Prem, Kumkum, Sandhya, Neha and Sumeet. Finally, I owe thanks to Meera Saxena (without whom etc. - but it's really true).

This research was funded by the State of Texas through TATP Project 003658-237 and the IBM Corporation through grant 61653.

Ravi Kumar Jain

The University of Texas at Austin

December, 1992

SCHEDULING DATA TRANSFERS IN PARALLEL COMPUTERS AND COMMUNICATIONS SYSTEMS

Publication No._____

Ravi Kumar Jain, Ph.D.

The University of Texas at Austin, 1992

Supervisors: J. C. Browne, John S. Werth

The performance of many applications of parallel computers and communications systems is limited by the speed of data transfers rather than the speed of processing. An important, but neglected, aspect of resource management to overcome this bottleneck is the scheduling of data transfers. Data transfer scheduling differs from traditional scheduling problems in that data transfer tasks require multiple resources simultaneously, rather than a single resource serially, in order to execute.

We study the data transfer scheduling problem by first defining a general model for precisely specifying and classifying scheduling problems. We use the model for the recognition of the similarity of seemingly different problems

from different application areas, for the systematic transformation of one problem specification into that of a seemingly different problem, and for the systematic decomposition of a problem specification into solvable subproblems.

We obtain polynomial-time, optimal and approximate algorithms for a wide range of data transfer scheduling problems under a variety of architectural and logical constraints, including communication architectures in which resources are fully connected, communication architectures with a tree topology, and the presence of mutual exclusion and precedence constraints. Our algorithms either generalize previous results for these problems, or provide better performance, or both.

Our results are applicable to both parallel computers and communications systems, including certain types of shared-bus multiprocessor systems such as the Sequent and the IBM RP3, hierarchical switching systems, tree-structured multiprocessor architectures, and intersatellite communications systems.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Extracting optimal performance continues to be a critical issue in computing and communications systems. Even as faster parallel computers and high-speed communications networks become available, newer applications such as image visualization and real-time databases stretch the limits of their performance. Improvements in underlying technology alone are insufficient to keep pace with these increasing demands. Sophisticated management of the resources provided by cheaper technology is required. An important component of resource management is scheduling, and in particular the scheduling of data transfers.

In this dissertation we study the scheduling of data transfers in parallel computers and communications systems. We obtain a general model for specifying scheduling problems which allows us to identify useful results in different application areas, and extend and apply them across application areas. We obtain optimal and approximate algorithms for a wide range of data transfer scheduling problems under a variety of architectural and logical constraints.

We focus on data transfer scheduling because this component of resource

management has long received insufficient attention in the area of parallel computer systems. For many applications it is not the processors but the data transfers for input/output (I/O) that are the bottleneck in parallel computer system performance. The continuing increase in computing speed relative to the speed of I/O devices, and the increasing I/O demands of new applications, indicate that the I/O bottleneck will be even more serious in the future. Parallel computer systems will not fully realize their potential performance, unless not only the computation but the I/O is performed in parallel, and equally importantly, unless the I/O resources and parallel I/O tasks are managed efficiently. However, while the scheduling of multiple processors has been studied extensively, there has been almost no study of scheduling parallel I/O tasks.

For communications systems, on the other hand, particularly satellite switching systems, scheduling data transfers has been studied for over a decade. Since the mid-80's, research on scheduling file transfers in computer networks has also been pursued. Nonetheless, improvements continue to be necessary as satellite and computer communication networks become ubiquitous, ambitious satellite networks using intersatellite communications links become operational, and new applications generate increasing data transfer demands.

Previous work done on scheduling over the last several decades in the fields of operations research, management science, and engineering, is typically not applicable to the problem of scheduling data transfers. The reason is that data transfer tasks require multiple resources simultaneously, rather than a single resource serially, in order to execute. Almost all previous work on scheduling theory has concentrated on the single-resource-per-task situation.

Thus extracting optimal performance requires abandoning traditional scheduling techniques and developing new algorithms and heuristics that perform simultaneous resource scheduling. The techniques that have been developed for data transfer scheduling in communications systems are an exception; almost all other work has concentrated on studying single resource scheduling problems such as job-shop scheduling, flow-schop scheduling, and the like.

It is typical of the fragmentation of the scheduling literature that previous research on data transfer scheduling for satellite communications has been performed and reported using specialized notation, jargon and narrow assumptions which hold only in the context of that application. The situation is further exacerbated by the explosion in the quantity of published research on scheduling since the 1950s. The consequence is that it is difficult to recognize when a scheduling problem in one application area has already been studied for a different application, let alone transfer research results across application areas. Thus, for instance, the results on scheduling data transfers in communications systems have not previously been applied to other applications such as scheduling parallel I/O, even though they provide potentially useful techniques and insights. What is required is a means of specifying and classifying scheduling problems, as well as solution techniques and algorithms, in a uniform abstract framework that exposes the underlying similarity of scheduling problems in diverse application areas.

To summarize thus far, we have outlined four motivating issues. Firstly, the scheduling of data transfers is a neglected but increasingly important component of resource management for extracting optimal performance from parallel computer systems. Secondly, while data transfer scheduling has received

some attention for communications systems, increasingly complex systems and applications demand improved techniques. Thirdly, previous results on scheduling do not apply to data transfer tasks since data transfers require multiple resources simultaneously in order to execute. Finally, a uniform abstract framework is required for specifying scheduling problems and their solutions so that the benefits of research results can be transferred across application areas. In this dissertation we address all four issues.

In the rest of this chapter we discuss these issues in more detail, outline our research approach, specify the specific problems that we will attack, and summarize the results we have obtained as well as suggestions for future work.

1.1 The parallel I/O bottleneck

We discuss the parallel I/O bottleneck in detail in this section. We first give a historical perspective. We then motivate the use of parallel I/O scheduling as an important addition to the solutions being developed to address the parallel I/O bottleneck.

1.1.1 Historical perspective

It has long been recognized that a memory hierarchy is required in order to satisfy the data requests of a CPU, and that the mechanical delays associated with input output (I/O) devices represent a significant potential

bottleneck in computer system performance (see Gibson [59] for a historical review). Indeed, the introduction of multiprogramming in computer systems was motivated by the need to overcome the sequential I/O bottleneck [26]. Nonetheless, "Input/Output has been the orphan of computer architecture" [68], and the I/O subsytem has received disproportionately little attention in sequential computer system design.

The I/O subsystem has received even less attention in the design of parallel computer systems. However, since the early 80's there has been a growing awareness of the I/O bottleneck in parallel systems [12, 16, 17]. It was in the early 80's that the performance of database machines designed in the late 70's (e.g. DIRECT [35]) were found to be severely constrained by I/O bandwidth [12]. While the I/O bottleneck remains a central concern in database machine architecture today [36, 13, 117], in recent years the concern has spread to general purpose supercomputers [131, 59, 103, 18] as well as mid-range and low-end machines [1]. As a case in point, while the early hypercube computers neglected the I/O subsystem, there have recently been many efforts to address the I/O bottleneck in hypercube system [14, 65, 116, 118, 121, 44, 57]. In fact, it has recently been argued that the data transfer capabilities of a system, including its I/O capabilities, should replace processing speed as the fundamental performance metric. For instance, Smith et al [131] predict that "The performance of supercomputers will ultimately be measured by how fast they can move data both within the system and across the network". Similarly, Jordan argues that for high performance systems, instead of the peak floating point rate, "A better measure of computer performance is data transport capacity" [84]. In addition, within the last year the parallel I/O bottleneck has been receiving substantial attention in the industry and general professional

literature (e.g. [67, 104, 18])

1.1.2 Scheduling parallel I/O

There are three basic reasons for the existence of the parallel I/O bottleneck. The first is the increasing discrepancy between the speed of computation and the speed of I/O. The second is the dramatic increase in the data demands of new applications such as image visualization and real-time databases. The third is the inability of dynamic RAM, despite its spectacular advances, to replace secondary storage devices. (See Gibson [59] for a detailed discussion of these issues).

Several attempts have been made to address the parallel I/O bottleneck in the last few years. They include approaches such as decreasing the number of I/O requests (improved or larger caches, larger block sizes, improved data allocation to allow contiguous files), increasing the parallelism of I/O requests (overlapping I/O with computation where possible, using asynchronous I/O), decreasing average disk access times (reducing utilization, introducing buffering, scheduling requests that are waiting at the disk controller), special I/O devices and controllers (multiple I/O processors, optical disks), and replacing secondary storage by RAM (or optical RAM). As argued by Gibson [59], none of these approaches provides a general-purpose solution that is satisfactory.

An important class of new approaches has been the introduction of synchronized disk interleaving [88], and disk striping [124] or data declustering [97].

The disk array approach [82], in particular Redundant Arrays of Independent Disks (RAID), combines the benefits of these approaches with increasing the ratio of disk heads to user data [113, 59].

For applications with high data demands for entities (e.g. files) of known size stored at known locations, and where the demand is relatively irregular, scheduling parallel I/O operations is an attractive addition to the set of techniques available for attacking the parallel I/O bottleneck. While the scheduling of I/O operations has been studied in the context of sequential computer organizations [33], the potential for improving parallel system performance by scheduling parallel I/O operations has been almost completely neglected.

As discussed below, review of previous work reveals that almost all previous research on parallel scheduling deals with tasks which require only a single resource at any given time. Single resource scheduling is not relevant for parallel scheduling of I/O operations where each operation requires multiple resources (e.g. processor, transfer media and external memory) in order to execute. Serial acquistion of multiple resources does not in general lead to optimal schedules; algorithms which simultaneously assign multiple resources to the members of a request set are required. This dissertation focuses on algorithms appropriate for centralized scheduling of batched I/O operations.

1.1.3 The structure of I/O requests

The conditions for scheduling to be effective are that there are choices to be made which affect performance, and that there are resource bottlenecks whose utilization can be improved by scheduling. The nature of the stream of I/O requests and the resource configuration determine whether these conditions hold, and if so, the characteristics of scheduling algorithms that will be effective.

The stream of I/O requests generated by multiprogrammed workloads is largely uncorrelated and has traditionally been scheduled dynamically at the level of device controllers or channels. There is usually sufficient randomness among requests to avoid long queues at any given disk so that scheduling parallel I/O operations above the controller level is of little benefit. However, the amount of correlation among I/O requests varies substantially with the application. Certain applications, such as 3D migration codes in seismic processing where the solution progresses systematically across a coordinate space, yield highly structured and totally predictable patterns of requests for data. In this case scheduling of I/O requests is of little benefit since the order of the requests can be predicted in advance and thus the problem reduces to one of assigning data to storage devices so as to minimize conflicts [17, 85]. On the other hand, certain families of applications, such as 3D visualization and decision support systems, pass through phases with different degrees of parallelism in computation and I/O requests. These applications, whether executing in parallel or sequentially, generate I/O requests in bursts as the locality of the data to be displayed or analyzed changes. It is often the case that the entire set of requests must be satisfied before the computation can

proceed. These families of applications may benefit substantially from batch-oriented scheduling of parallel I/O operations if resource bottlenecks exist in the I/O system architecture.

As far as resource bottlenecks go, it is typically the case in large-scale parallel architectures that there are fewer access paths to I/O devices than either I/O devices or processors. This is commonly observed in current bus-oriented architectures (e.g. [100]).

The combination of bursts of I/O requests and potential resource bottlenecks suggest that there may be utility in efficient algorithms for generating parallel schedules. Efficient algorithms are needed because they will be executed repeatedly during the execution of the applications. This dissertation develops and characterizes algorithms which are applicable to a significant class of I/O system architectures.

1.2 Data transfers in communications systems

In contrast with parallel computer systems, the desirability of scheduling data transfers in communications systems has long been observed and accepted. There are two major applications areas where it has been actively pursued: satellite communications and computer networks.

The first operational communications satellite was deployed in 1965. Some of the earliest work on data transfer scheduling dates to the late 70's [75, 87, 38].

In satellite switching systems, the motivation for scheduling has arisen from a need to maximize utilization of switch hardware, or minimize the number of times that the switch has to be reconfigured for a given set of input transfers, or to minimize the probability that an incoming transfer is blocked because either an input port or an output port of a switch is unavailable. In time division multiple access (TDMA) satellite switches, this problem is known as the time slot assignment problem. There is a substantial literature on various aspects and approaches to this problem [75, 74, 9, 10, 7, 39, 111, 66, 11, 53, 96, 19, 20, 54, 21, 22, 81, 77, 125, 126, 136, and references therein] which will be reviewed at the end of each relevant chapter as necessary.

Another area where scheduling data transfers in communications systems arises is the scheduling of file transfers in a computer network. The earliest work on this dates to the mid-80's, starting with Coffman et al's landmark paper [27]. Typically the objective here is to minimize the total amount of time that the transfers take to complete. In contrast to the work on satellite switching systems, research has tended to consider general communications topologies, non-preemptive transfers, and diverse kinds of communications devices, such as transceivers. Nonetheless, the underlying issues in the two application areas are very similar; once again it is symptomatic of the methodology in scheduling research that there is almost no recognition of this fact and no attempt to exploit it. The literature on this problem is not as large as for satellite switching - perhaps because of the remarkable number of results already contained in the original Coffman et al paper - but it is still significant [27, 23, 24, 25, 66, 142, 77, 125, 101]. It will be reviewed at the end of each relevant chapter as necessary.

While there has been substantial work on data transfer scheduling in communications networks, improvements continue to be necessary. Most of the scheduling algorithms have high time complexities that make them impractical, particularly for the satellite communications application. Moreover, ambitious satellite communications systems which were only being discussed in the research literature as little as a decade ago, such as satellites connected by intersatellite links [87], are now being designed. These communications systems present a host of new challenges and constraints that alter the satellite switching environment considerably. Finally, for both satellite systems and terrestrial computer networks, new applications such as scientific visualization [32, 110, 2], videoconferencing and multimedia information systems [139, 43, 90, 120], and personal communications services [99, 73] are creating tremendous demands on the available communications resources.

In this dissertation, we will explicitly address the concerns of obtaining faster scheduling algorithms and heuristics, covering more sophisticated communications architectures such as networks using intersatellite links, and applications such as 3D visualization of scientific data stored in image databases.

1.3 Simultaneous resource scheduling

One of the reasons that there is little previous work that applies to scheduling data transfers is that the problem involves simultaneous resource scheduling. The vast majority of previous work on scheduling resources concerns scheduling tasks which require a single resource at a time. In computer science, this includes the tremendous amount of research on disk, drum and CPU

scheduling carried out in the late 60's and 70's [129]. It also includes most of the research on scheduling tasks on multiprocessors carried out since then; this literature is reviewed in Chapter 2.

Outside the context of communications systems described previously, to our knowledge there is no research on scheduling multiple resources simultaneously that is relevant to our concerns. The research carried out in the context of management [130, and references therein], operations research [98], manufacturing [3, and references therein], multiprocessing computer systems [55, 56, and references therein], and real-time multiprocessing computer systems [146, 128, and references therein] is interesting but of limited usefulness to us. The primary reason for this is that in these papers the simultaneous resource requirements have been addressed in a very general fashion, leading immediately to problems that are known to be NP-complete [56] or which require general linear programming solutions of unacceptably high time complexity [130]. In contrast, we seek to exploit the special structure of simultaneous resource requirements that arise in data transfer tasks, and hence derive polynomial-time algorithms and simple, fast heuristics that are effective for our application. In later chapters we will demonstrate the results of this approach.

1.4 Statement of the problem

The fundamental problem studied in this dissertation is the scheduling of data transfers in parallel computers and communications systems, with special reference to parallel I/O, and satellite and computer communications

applications. We thus investigate an important special case of the problem of scheduling tasks which require multiple resources simultaneously. We also investigate the problem of identifying and exploiting the underlying similarity of scheduling problems drawn from different application areas.

The simplest case of the data transfer problem we study, which we call Simple Data Transfer Scheduling or $SimpleDTS$, can be stated informally as follows:

Given a set of data transfers, where

1. each transfer requires a fixed but possibly distinct time,
2. each transfer requires a specified pair of resources, one from each of two given sets of resources,
3. each resource belonging to one resource set can communicate via a direct dedicated link with every resource in the other set, and
4. the transfers may occur in any order,

is there a preemptive schedule for performing the transfers whose total length is at most some given bound?

When the problem is stated as an optimization problem the objective is to minimize the schedule length. An example is given below.

Example. An instance of the $SimpleDTS$ scheduling problem is given for the parallel I/O application in Fig. 1.1. Assume that each processor and each

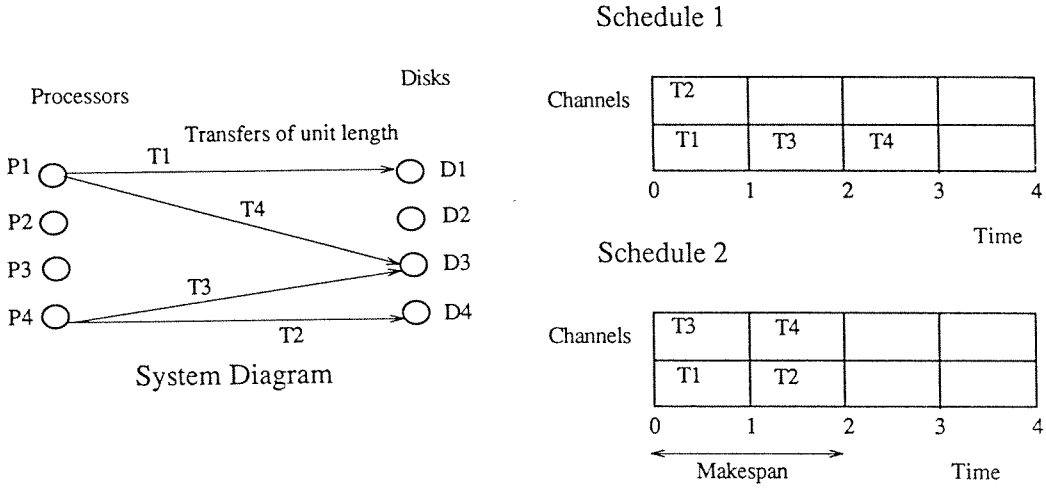

Figure 1.1: Parallel I/O scheduling example

I/O device can participate in at most one data transfer at any given time, and each transfer is of unit length. Clearly the minimum length of time for completing the transfers corresponds to the optimal schedule rather than the schedule obtained by executing the tasks in the order they are numbered.

In this dissertation we will consider the following specific problems, each of which has applicability to both parallel I/O scheduling as well as scheduling data transfers in communications systems. The problems are necessarily described informally at this stage; they will be specified formally in the dissertation.

1. The development of a general abstract model for specifying scheduling problems.
2. The problem of developing optimal and approximate algorithms for solving the $SimpleDTS$ problem.

3. The problem of developing optimal and approximate algorithms for the *SimpleDTS* problem where the communication architecture restricts the number of simultaneous transfers possible.
4. The problem of developing optimal and approximate algorithms for preemptive data transfer scheduling where the communication architecture has a tree topology ('tree architectures').
5. The problem of developing algorithms for preemptive data transfer scheduling in tree architectures where preemptions may occur at non-integer boundaries, and both 'local' and 'remote' data transfers may take place.
6. The problem of developing algorithms for scheduling data transfers in the presence of mutual exclusion constraints and precedence constraints.
7. The problem of experimentally evaluating the effectiveness of algorithms for *SimpleDTS* and for scheduling in tree architectures.

1.5 Research approach

We first approach the problem of defining an abstract model for specifying scheduling problems by using a formal graph-theoretic formulation. We choose a graph-theoretic approach because graph theory has proven useful as a unifying modeling and analysis formalism for a diverse range of applications [34], and because of its wide accessibility.

The scheduling model is not only a research result but an invaluable aid in our approach to the other problems we study. We formally specify all the

problems in the framework of the model, allowing us to easily expose the underlying similarities of problems in different application domains.

The graph-theoretic nature of the problem specifications in our model immediately points to fundamental computational graph theory techniques for solutions. In particular, we are able to apply and extend graph matching, edge coloring, and network flow techniques for developing solutions to our problems. We can do this with confidence because the formal nature of the problem specifications eliminates any ambiguities or doubts about the applicability of these techniques.

The scheduling model also allows us to manipulate the problem specifications. This has tremendous practical benefit in some cases, as we are able to avoid developing new scheduling algorithms for new problems, by exploiting existing solutions. We use the model to *recognize* the equivalence of scheduling problems drawn from different applications by inspecting their specifications in the model. We also use the model to systematically *transform* a problem specification into the specification of a seemingly different problem. And finally we show that it is possible to *decompose* problem specifications into sub-problems whose solution is known, and solve the original problem by combining these solutions.

An important aspect of our research approach is to experimentally evaluate our scheduling algorithms. While doing so it is important to carefully consider and state the assumptions and the range of parameter values for which the algorithms are to be evaluated. In order to do so effectively, it is desirable

to select specific application scenarios and operating conditions. We have focused on the parallel I/O application, and in particular, for the projected workloads from high-demand applications such as 3D visualization of scientific data.

1.6 Summary of results obtained

We obtain the following results:

1. An abstract graph-theoretic model for specifying scheduling problems, and a demonstration of its use for specifying a wide range of traditional and simultaneous resource scheduling problems [80, 81].
2. A set of three optimal algorithms, **A1 - A3**, for solving the *SimpleDTS* problem, and for *SimpleDTS* where the architecture restricts the number of simultaneous transfers [78]. These algorithms generalize previous work for these problems and provide algorithms with better time complexity.
3. A detailed experimental evaluation of the performance of **A1 - A3** for the parallel I/O application, with a determination of the situations for which each is best suited.
4. A theoretical analysis of the worst-case time complexity and schedule length generated by two heuristics for the situations covered by **A1 - A3**, when all tasks are of the same length. We prove that the heuristics produce schedules that are at most twice the length of the optimal schedule.

5. An optimal algorithm, **Tree**, for preemptive data transfer scheduling where the communication architecture is a tree [77]. This algorithm solves more general cases of this class of scheduling problems than previously available algorithms, and provides a better time complexity.
6. An extension of the **Tree** algorithm to optimally solve problems where preemptions may take place at non-integer boundaries, reflecting the characteristics of multimedia applications involving continuous media [125, 126].
7. An approximate algorithm for preemptive data transfer scheduling in tree architectures.
8. An approximate algorithm for scheduling data transfers in tree architectures when both local and remote data transfers may take place [81].
9. An optimal algorithm for scheduling data transfers where there are no architecture constraints but the tasks are instead subject to logical mutual exclusion constraints [81]. The allowable class of mutual exclusion constraints includes those expressible in the CODE 1.2 parallel programming environment [140].
10. The data transfer problem is NP-complete if precedence constraints are permitted, even if their structure is restricted to be a tree.

1.7 Organization of the thesis

In Chapter 2 we define our model for specifying scheduling problems. In the following chapters we consider specific scheduling problems and their solutions. We give a survey of previous related work and suggestions for future

work as needed in each chapter. In Chapter 3 we design solutions to the first problems we consider, *SimpleDTS* as well as *SimpleDTS* when only a restricted number of transfers may occur in parallel. We present the results of an extensive experimental evaluation of the optimal solution algorithms. In Chapter 4 we discuss heuristics that have been proposed and experimentally evaluated for these problems, but for which no analysis of time complexity or worst-case schedule length was previously given; we derive both. In Chapter 5 we consider data transfers in hierarchical, tree-structured architectures, and design an optimal algorithm which generalizes previous work for this problem and obtains a better time complexity. In the following chapter we discuss an approximation algorithm for this problem. In Chapter 7 we consider various extensions and applications of the algorithm for tree architectures, including arbitrary preemptions and both local and remote data transfers. In Chapter 8 we consider the effects of mutual exclusion and precedence constraints, and in Chapter 9 we end with some conclusions and some broad suggestions for future work.

Chapter 2

A Model for the Scheduling Problem

We define our scheduling model in this chapter. The model restricts attention to providing a framework for formally specifying scheduling problems which are static and deterministic, i.e., all relevant problem parameters are fixed and known *a priori*.

The term "scheduling" has sometimes been used loosely in the literature, with different interpretations in different application domains. In the following we define the terms allocation, assignment and schedule precisely. We later give examples of practical problems drawn from different application areas where the object is to calculate an allocation, an assignment or a schedule for a given set of tasks. Finally we compare the model with previous classification schemes.

2.1 Basic Definitions

We take certain notions as primitive. We shall assume the existence of primitive objects called *resources*; intuitively these correspond to machines, parts,

communications links, disks etc. We also assume the existence of primitive objects called *units of computation*; intuitively these correspond to computer subroutines, data transfers, industrial processes, etc. Finally, we assume discrete *time* to be a primitive notion, represented in the model as the set of natural numbers. The following two definitions relate these primitive notions.

Def. A *task* is a unit of computation that requires a fixed set of resources.

Notice that this definition does not assume that the actual resources required by a task are known, only that they form a fixed set. It is assumed that all the resources are required for the entire duration of the task. In practice a complex process requiring different sets of resources at different times may be represented as a sequence of tasks.

Def. The *length* of a task is the amount of time the task requires its fixed set of resources.

In practice we may specify the task length in units such as machine instructions or machine cycles, from which the task length in units of time can be readily calculated knowing the speed of the resource.

We assume that resources are partitioned into a collection of disjoint sets; the set to which a resource belongs is called its *type*.

Notation. Let T denote the set of tasks, R the set of resources, and RT the set of resource types. Let τ denote the set (of natural numbers) representing time and N the set of natural numbers.

Def. The *task resource requirement* is a function $tr : T \rightarrow N^{|RT|}$ specifying for each task the number and type of resources required by the task.

We assume that the task resource requirement is known for all tasks, i.e., is an "input" to any problem we shall consider. The solutions of different types of problems are the allocation, assignment and schedule functions, which we define as follows.

Def. An *allocation*, $al : \{T\} \rightarrow N^{|RT|}$, specifies the number of resources of each type to be used by the set of tasks T.

An allocation differs from a task resource requirement in that it refers to the number and type of resources used by the *set* of tasks as a whole. For instance, 5 independent tasks, each requiring a tape drive, may be allocated only a single tape drive and hence have to be serviced in sequence.

Def. An *assignment* is a function $as : T \rightarrow 2^R$ specifying the resources to be used by each task.

An assignment differs from an allocation in that it specifies, for each task, the exact resource instance that the task requires. For example, given 5 tasks $T_1, ..., T_5$, specifying that task T_i requires tape drive unit (i **mod** 3) is an assignment.

Def. A *schedule* is a function $s : T \rightarrow 2^R \times 2^{\tau \times N}$ specifying for each task the resources and the times τ_i and durations n_i for which they are held, i.e., for each task $t \in T$, there exists $k \in N$ such that

$$s(t) = R(t) \cup \{(\tau_1, n_1), (\tau_2, n_2), ..., (\tau_k, n_k)\}$$

where, for $1 \leq i \leq k$, $R(t) \subseteq R$, $\tau_i \in \tau$, $\tau_{i+1} \geq \tau_i + n_i$, and $n_i \in N$.

Note that if schedule s is non-preemptive then for all $t \in T, k = 1$, i.e., there is only a single non-interrupted block of contiguous time slots during which the task executes.

Def. Given a schedule $s(t)$ as defined above we can define the following auxiliary functions. Functions *start* and *stop* give the times at which a given job starts executing and is completed, respectively. The *makespan* is the time from the start of the earliest job to the time that the latest job completes. Function *active* gives the time slots during which a given job executes.

$$start(s,t) = \tau_1$$
$$stop(s,t) = \tau_k + n_k$$
$$makespan(s) = max\{stop(s,t) : t \in T\} - min\{start(s,t) : t \in T\}$$
$$active(s,t) = \{(\tau_i, \tau_i + 1) : \exists\, j, 1 \leq j \leq k, \tau_j \leq \tau_i \wedge \tau_i + 1 \leq \tau_j + n_j\}$$

2.2 Allocation and Assignment Problems

We show how allocation and assignment problems are defined in the model. The definitions introduced here are used to define the scheduling problem.

Assuming a task resource requirement is given, the problem of computing an allocation meeting specified constraints among the tasks and minimizing some

objective function is called the allocation problem. The constraints among the tasks are specified using an *extended precedence graph*, defined below.

Terminology. A hyperedge is an undirected connection between one or more vertices. A hyperedge on one vertex is called a self-loop. A hyperedge on two vertices is called an edge. (We represent hyperedges connecting three or more vertices as a line incident on the vertices). An arc is a directed edge. A path is a sequence of arcs or hyperedges in which consecutive arcs or hyperedges share a vertex and no vertex is included twice. A (linear) chain is a path consisting only of arcs. A cycle consists of a path and an arc or hyperedge connecting the first and last vertex of the path.

Def. An *extended precedence graph* $PG = (T, Ep, Lp)$ consists of a set T of vertices representing tasks, a set Ep of arcs and hyperedges, and a labeling function Lp where

$$Lp(x) = \begin{cases} \text{length of task } x, \text{ if } x \in T \\ \text{communication cost from task } u \text{ to } v, \text{ if } x = (u, v) \text{ is an arc} \\ 0, \text{ if } x \text{ is a hyperedge} \end{cases}$$

Informally, a hyperedge specifies that the tasks connected by the edge are to be mutually exclusive, i.e., no two may execute or operate simultaneously. An arc (u, v) specifies that task u must complete before task v may begin. These notions are made precise when the scheduling problem is defined below. We will sometimes refer to the extended precedence graph simply as a precedence graph for brevity.

Def. An *allocation problem* is a tuple $ALP = (PG, f)$ where f is an objective function and PG is a precedence graph.

Example 1. Consider a set of 5 tasks, $T_1, ..., T_5$, each of which consists of a unit-weight data transfer between a processor and a disk. If the tasks are independent, PG consists of 5 vertices and no arcs. If the the objective is to minimize the number of disks and processors so as to achieve a minimum makespan, upto 5 processors and disks can be used in parallel. However, if T_1 is to precede T_2 which is to precede T_3, PG consists of 3 vertices labeled T_1, T_2, T_3 with an arc from T_1 to T_2 and an arc from T_2 to T_3, and two vertices labeled T_4, T_5 with no arcs incident upon them. In this case, only 3 processors and 3 disks need to be allocated to obtain a minimum-length schedule.

A precedence graph with no hyperedges, i.e., consisting only of arcs, is a familiar structure from previous work in computer science areas such as parallel architectures, compilers, etc., as well as traditional scheduling theory. The addition of hyperedges is necessary to express the synchronization requirements between parallel tasks commonly encountered in parallel programming. In our model the precedence graph is used also to express what are known in scheduling theory as *technological constraints* [45], such as the sequence in which jobs visit machines in a job-shop. Finally, the vertices in the precedence graph can be annotated with additional task information, such as release times, deadlines, etc., although these will not be considered in this paper.

Assuming a task resource requirement is given, the process of computing an assignment meeting specified constraints among the resources as well as the tasks and minimizing some objective function is called the assignment problem. The constraints among the resources are specified using an *architecture graph*, and are called *architecture constraints*.

Def. An *architecture graph* $AG = (R, Ea, La)$ consists of a set R of vertices representing resources, a set Ea of arcs and hyperedges, and a labeling function La:

$$La(x) = \begin{cases} \text{processing speed, if } x \in R \\ \text{capacity, if } x \in Ea \end{cases}$$

An arc or hyperedge in AG represents interconnection of resources. Typically hyperedges represent buses and arcs represent unidirectional communication links.

Def. An *assignment problem* is a tuple *ASP = (PG, AG, f)* where f is an objective function, PG is a precedence graph, and AG is an architecture graph.

Example 2. Consider the set of 5 unit-length tasks $T_1, ..., T_5$, with T_1 to precede T_2 which is to precede T_3, as in Example 1. Suppose AG consists of 5 processors and 5 disks interconnected such that there is a direct dedicated link between every processor-disk pair. The objective function is to obtain a minimum-length schedule while utilizing a minimum number of resources. Since at most three of the tasks can be active at any given time, only 3 processors and 3 disks need to be allocated. Further, any two tasks can be assigned the same processor-disk pair if there is an arc connecting them in PG.

We now give examples of two applications in computer science and engineering that demonstrate the ability to specify realistic allocation and assignment problems using the model.

Application 1: Digital Hardware Synthesis. A common problem in the manufacture of application-specific integrated circuits (ASIC) is to design a VLSI chip that implements a given computation, e.g. a linear filter for signal processing. The computation can be broken down into a set of tasks, e.g. FFT, multiplication, etc., that are to be performed in some specific partial order, where each task requires a known set of resources (adders, invertors, etc.). Depending on the computation, it may be possible to reuse some of the resources for different tasks, e.g. two tasks that each require an adder may be able to time-share a single physical adder circuit. Then a typical allocation problem is to determine the number of resources of each type that are required in order to perform the computation such that the cost is minimized; the cost may be simply the total number of resources, or the chip area required to implement them, etc. [114]. The problem can be specified in the model as a precedence graph representing the computation and an objective function which calculates the cost of the number and type of resources used.

Application 2: Parallel Programming. A well-known problem is to minimize the execution time of a parallel program on a given parallel computer architecture, where the program has been decomposed into a set of parallel tasks. One objective is to allow the parallel tasks to execute on as many separate processors as possible so as to reduce computation time. This conflicts with the objective of clustering tasks on a single processor to minimize the delay incurred when data is communicated among them. (See, for instance, [89]). In our model the problem is an assignment problem in which the parallel program is specified as the set of tasks in a precedence graph and the given computer architecture as an architecture graph; the objective function is typically the makespan.

Note that in the digital hardware synthesis problem, once an allocation has been calculated, the assignment problem is often trivial. The precedence graph and allocation together determine the form of the architecture that is to be synthesized, and hence the assignment of tasks to resources in that architecture.

2.3 Definition of the Scheduling Problem

Unlike the allocation and assignment problems, the scheduling problem is directly concerned with determining the times at which tasks execute. In order to define the scheduling problem, we first introduce the *resource graph*. A resource graph specifies the assignment, if it is known, as well as the direction of any data transfers that are to take place between resources that are held simultaneously.

Def. A *resource graph* for a given set of tasks T, $RG = (R, Er, Lr)$, consists of the set R of vertices representing resources, a set Er of arcs and hyperedges, and a labeling function Lr. For all arcs and hyperedges $e \in Er, Lr(e) = t$ specifies that task $t \in T$ must simultaneously possess all resources connected by e. In addition, if $e = (r, s) \in Er$ is an arc then t involves transfer of information from r to s. If $Er = \{\}$ then the assignment of tasks to resources is not specified.

In some applications, such as the the hardware synthesis and parallel programming examples given above, there may be explicit allocation and assignment phases that occur before the scheduling phase, so that the assignment

is known before scheduling is begun. In other applications, however, such as multiprocessor scheduling (i.e., scheduling identical parallel machines) the process of computing an assignment is performed at the same time as that of scheduling. In the latter class of problems only the task resource requirement is known; typically this occurs because there is only one resource type and the interconnection of resources is nonexistent or trivial.

Def. A *scheduling problem* is a tuple $SP = (PG, AG, RG, f, Preempt)$ specifying constraints on tasks and resources, where f is an objective function, $Preempt$ is true iff the schedule may be preemptive, and PG, AG and RG are precedence, architecture and resource graphs respectively.

The resource graph, whether specified as part of the problem or calculated during the scheduling process, must be "consistent" with the architecture graph, since both refer to resources and data transfer between them. To capture this requirement we define a resource function.

Def. A resource function $g : RG \rightarrow AG$ is a function which

1. if $e = (u, v) \in Er$ is an arc in RG then there is a path from $g(u)$ to $g(v)$ in AG
2. ignores all hyperedges and labels in RG.

In some cases the resource function may be very simple. For instance, if the architecture graph contains only arcs and is complete, and there are no hyperedges or parallel arcs in the resource graph, the resource graph will

simply be a subgraph of the architecture graph. Also, for brevity we may not specify a resource graph completely. For instance, if the architecture provides a unique directed path between every resource pair, the resource graph may only specify the end vertices involved in a data transfer, leaving the intermediate vertices along the data path implicit. We will use such shorthand notation in some of the examples in this paper.

In the following definitions we formalize the notion of a schedule being a solution to a scheduling problem posed in the model. Most of the definitions are obvious from the semantics of the various graphs that we have defined. We include the resource function in the definition of a schedule satisfying a problem in order to incorporate a notion of consistency.

Def. A schedule s satisfies the precedence graph $PG = (T, Ep, Lp)$ of a scheduling problem *SP = (PG, AG, RG, f, Preempt)* iff

1. If $(u, v) \in Ep$ is an arc in PG then task u stops before task v begins, i.e., $stop(s, u) \leq start(s, v)$
2. If $e \in Ep$ is a hyperedge in PG then no two tasks connected by the hyperedge are active simultaneously, i.e.,
   $\forall\, u, v \in e, active(s, u) \cap active(s, v) = \{\}$
3. For all $t \in T$, $Lp(t) = |active(s, t)|$.

Def. A schedule s satisfies the architecture graph $AG = (R, Ea, La)$ of a scheduling problem *SP = (PG, AG, RG, f, Preempt)* iff

1. For all $e \in Ea$, for all i, $0 \leq i \leq$ *makespan*(s), if $a(i)$ is the number of tasks active during time slot i which use e, then $a(i) \leq La(e)$.

Def. A schedule s satisfies the resource graph $RG = (R, Er, Lr)$ of a scheduling problem *SP = (PG, AG, RG, f, Preempt)* iff

1. For all edges $e \in Er$, if $r(e)$ is the set of resources connected by e, $r(e) \subseteq RLr(e))$

Def. A schedule s satisfies a scheduling problem *SP = (PG, AG, RG, f, Preempt)* iff

1. s satisfies PG, AG, and RG
2. If $Preempt = false$ then for all $t \in T$, for some $\tau' \in \tau, n \in N$, $s(t) = R(t) \cup (\tau, n)$
3. there exists a resource function $g : RG \rightarrow AG$.

Def. A schedule is an *optimal* solution to a scheduling problem *SP = (PG, AG, RG, f, Preempt)* if it satisfies SP and the objective function f is minimized.

Application 1: Job-shop scheduling. We use the n-job, m-machine job-shop problem to give an example of how a traditional class of scheduling problems, not involving simultaneous resource requirements, can be specified formally in the model. In later sections we will specify problems relating to simultaneous resource scheduling, in particular parallel I/O scheduling.

$JobShop = (PG, AG, RG, f, Preempt)$

where $PG = (T, Ep, Lp)$ consists of n linear chains of m vertices each, each chain representing a job and each vertex an operation on a machine. Thus $|T| = n\ m$ and for all tasks $t \in T$, $Lp(t) \in N$ specifies the length of the task as a number of primitive *operations*.

$AG = (R, Ea, La)$ consists of $|R| = m$ vertices and no edges, i.e., $Ea = \{\}$. For all $r \in R$, $La(r) \in N$ specifies the processing speed in operations per unit time.

$RG = (R, Er, Lr)$ consists of $|R| = m$ vertices, each with n self-loops, i.e., $|Er| = n\ m$, and Lr is a bijection between Er and T. (The labels on the self-loops in Er together with the precedence order between tasks specified in PG determine the technological constraints, i.e., the order in which jobs visit machines in the job-shop).

Example 3. As a numerical example, consider a 3-machine, 3-job job-shop, where each job J_i visits each machine M_i in some specified order, for $i = 1, ..., 3$. The operation performed by job J_i at machine M_j is a task T_{ij}. The order in which jobs visit machines is shown in a "flow graph" in Fig. 2.1, followed by the specification in our model.

The model separates the specification of the technological constraints between tasks (specified in PG) from the interconnection of resources (AG) and from the specification of the resources that each task needs (RG). Since the machines are not interconnected, $Ea = \{\}$. Since each task requires exactly one

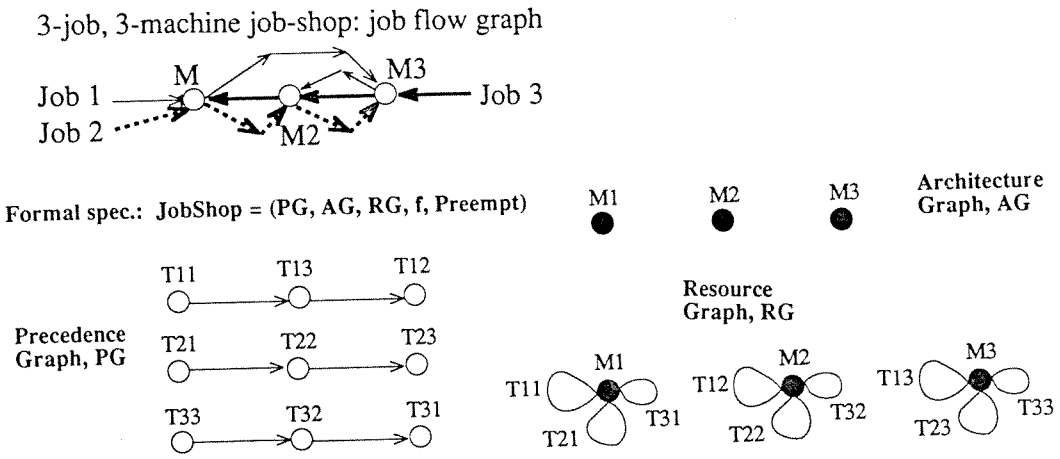

Figure 2.1: Specification of a job-shop example

resource, Er consists only of self loops. This separation of concerns in the problem specification is especially useful in order to manage the complexity of scheduling problems such as parallel I/O scheduling, as we'll see in later sections.

2.4 Example: Multiprocessor scheduling

In this section we show how the model is used to specify the multiprocessor (also called the uniform parallel machine) scheduling problem and classify results. By doing so we are able to survey a substantial portion of the literature on scheduling parallel tasks and recognize that it is not directly relevant to the problem of scheduling I/O operations. The multiprocessor scheduling problem has received a tremendous amount of attention (see [112], [93] and [60] for surveys). The basic problem is to schedule a set of n tasks with fixed and possibly unequal lengths and having a given precedence order, on a set of m identical processors, where each task can execute on any of the proces-

sors and the objective is to minimize makespan. In our model the problem is defined as follows.

$MS = (PG.AG, RG, f, Preempt)$

where
$PG = (T, Ep, Lp)$ where $|T| = n$, Ep is a set of arcs representing the precedence order. and $Lp(t)$ is the length for each task $t \in T$, $Lp(e) = 0$ for $e \in Ep$.

$AG = (R, Ea, L)$ where $|R| = m$, $|Ea| = 0$, and $La(r)$ is the (same) processing speed for each $r \in R$. Note $|RT| = 1$.

$RG = (R, Er, Lr)$ where $|Er| = \{\}$.

f is makespan.

The problem is referred to as non-preemptive or preemptive multiprocessor scheduling depending on the value of $Preempt$. Calculating the assignment is a significant part of the process of calculating a schedule.

We summarize previous work on optimal solutions to MS in Table 2.1 for the case where $Preempt = false$, and $Lp(t) = 1$ for all $t \in T$. In the table, $n = |T|$, $m = |Ep|$, "Arb." means "arbitrary", and $A(n)$ is the very slowly-growing inverse of Ackermann's function.

Ullman [134] shows that non-preemptive scheduling for tasks where $Lp(t) \in \{1, 2\}$ is NP-complete for precedence constraints consisting of arbitrary

DAGs and $m = 2$. Although approximation algorithms have been suggested for this case (see Lawler et al., [92]), the bulk of the research has concentrated on optimal algorithms for the tractable case where $Lp(t) = 1$, and is discussed here.

Hu's algorithm [72] can be viewed as a list scheduling algorithm in which jobs are entered in the list in accordance with their level in the precedence tree. It can also be viewed as a critical path method (CPM) scheduling algorithm, since the next job chosen is one on the critical path in the precedence tree. Gabow's algorithm [49] can also be viewed as a CPM algorithm.

The Coffman-Graham algorithm [28] is one of the best-known algorithms in this area, and runs in time $O(n^2)$ provided the input to the algorithm consists of a precedence graph whose transitive closure has been computed. The improvements by Gabow [49] and Gabow and Tarjan [52] remove this requirement, thus reducing the time complexity.

Goyal [64] considers the case where there are multiple resource types, but each task requires exactly one instance of exactly one type. In addition, there exists only one instance for each resource type. This can be mapped to the MS problem where $Lp(t) = 1$, and an assignment is supplied. Goyal shows the problem is NP-complete when PG is an arbitrary DAG or forest, and gives a linear-time scheduling algorithm for the case when PG is a "cyclic forest".

Palem [112] first shows that MS is a special case of another well-known problem, the precedence constrained minimum tardiness scheduling ($PCMTS$)

Reference	$\|R\|$	PG	Time Complexity
Hu, 1961 [72]	Arb.	forest	$O(n)$
Ullman, 1975 [134]	Arb.	DAG	NP-Complete
Fujii et al, 1969 [47]	2	DAG	$O(\min(mn, n^{2.61}) + n^{2.5})$
Coffman and Graham, 1972 [28]	2	DAG	$O(\min(mn, n^{2.61}) + m + nA(n))$
Gabow, 1982 [49]	2	DAG	$O(m + nA(n))$
Gabow and Tarjan, 1983 [52]	2	DAG	$O(m + n)$

Table 2.1: Previous work for non-preemptive multiprocessor scheduling.

problem. He then shows how several polynomially solvable cases of both MS and $PCMTS$ can be cast as a problem of finding an optimum sequence of edges in a hypergraph. The cases that can be shown equivalent using this framework include those described in this section, subcases of $PCMTS$, and cases of scheduling on pipelined processors. An algorithm to find the optimum sequence of hyperedges with the desired property can be performed in time $O(|T|^2 \log |T|)$.

Finally, Lenstra and Rinnooy Kan [95] show that the ε-approximation algorithm with the lowest worst-case polynomial time complexity has $\varepsilon = 4/3$, and Lam and Sethi [91] show that using the Coffman-Graham algorithm to generate lists gives approximation algorithms with $\varepsilon = 2 - 2/m$ for $m > 1$.

Previous work on MS where $Preempt = true$ is presented in Table 2.2. In the Table, "Arb." stands for "arbitrary" and "Mut. Com." stands for *mutually commensurable* task lengths. Two lengths are mutually commensurable if there exists a real number such that each length is some integer multiple of the real number. Lam and Sethi [91] develop a polynomial-time ε-approximation

Reference	$\|R\|$	$Lp(t)$	PG	Time Complexity
Ullman, 1976 [135]	Arb.	1	DAG	NP-complete
Muntz and Coffman, 1969 [108]	2	Mut. Com.	DAG	$O(n^2)$
Muntz and Coffman, 1970 [109]	Arb.	Mut. Com.	tree	$O(n^2)$
Gonzalez and Johnson, 1980 [61]	Arb.	Arb.	tree	$O(n \log m)$

Table 2.2: Previous work for preemptive multiprocessor scheduling.

algorithm based on Muntz and Coffman's method, and show that $\epsilon = 2 - 2/m$ for $m > 1$.

We observe that the multiprocessor scheduling problem essentially consists of scheduling a single resource per task under given precedence constraints, unlike the I/O scheduling problem, which requires two or more resources to be simultaneously accessed by each task. Typically these resources may be processors, disks, etc. In the following chapter we define parallel I/O scheduling precisely.

2.5 Comparison With Other Models

There have been several types of models proposed in the literature for the scheduling problem. In [3, 83] the Gantt chart is mentioned as a uniform model for representing a schedule once it has been computed, i.e., for the solution of a scheduling problem. In this section we discuss models for the specification of scheduling problems.

In [30] the well-known four-parameter notation $A/B/C/D$ is used to classify scheduling problems drawn mainly from the area of simple job-shop process scheduling. In [93] a classification is introduced consisting of a 7-tuple written as three fields $\alpha \mid \beta \mid \gamma$. It is assumed that there are n jobs to be processed on m machines, with at most one job per machine and at most one machine per job at any given time. The field α describes the machine environment (a single machine, identical parallel machines, flow-shop, etc.), β describes the job characteristics (preemption, precedence constraints, release times, upper bounds on number of operations per job, and processing times), and γ specifies the objective function. Elementary reductions among scheduling problems are described using this classification scheme.

In comparison with our model, the classification schemes of [30] and [93] have the advantage of compactness, but they are highly restricted in assuming that jobs require only a single instance of a single resource type at any given time. Hence they do not consider the interconnection of resources, as described by the architecture graph in our model. Thus the large section of the literature they address is not directly relevant to the parallel I/O scheduling problem. In addition the popular $A/B/C/D$ scheme is open-ended in nature, with the parameter C becoming longer as more complex problems need to be specified, so that it is not obvious if some problem constraint has been omitted inadvertently from the specification. The formal nature of our model facilitates complete and precise specifications of problem classes as well as individual problem instances. The model also divides the specification into modular sub-specifications (PG, AG, etc.), keeping the concerns of each sub-specification separate. It thus becomes possible to reason systematically about the relationships between problems, and to use well-defined and formal

manipulations on problem specifications.

We choose to use graph theory as the underlying formalism for several reasons. Firstly, graph theory has proven itself invaluable as a modeling and analysis formalism in a wide range of applications in areas as diverse as engineering, physical sciences, life sciences and sociology [34]. Secondly, fundamental graph theoretic problems, such as graph coloring and matching, have been found to underlie seemingly different problems in these areas, leading us to surmise that they may be useful for unifying scheduling problems drawn from different applications also. In later sections we see that this is indeed the case. Thirdly, the language of graph theory is intuitively appealing and accessible. Finally, graphs in some form are familiar to both theoreticians as well as practitioners in many different fields, particularly engineering and computer science, and are increasingly being taught and applied in these fields.

The graph-theoretic nature of our model lends itself to our suggestions for further work in this area. It would be useful to investigate whether the model can be further formalized to obtain 'meta-theorems' about the logical manipulation of problem specifications, including transformation, reducibility and equivalence of problem specifications. An example of the usefulness of such formalization can be found in the context of constraint satisfaction problems [123].

Chapter 3

Optimal Scheduling in Bus Architectures and TDM Switches

The specific scheduling problem to which the algorithms in this chapter apply is the following. Given a set of data transfers, where

1. each transfer requires a fixed but possibly distinct time,
2. each transfer requires a specified pair of resources, one from each of two given sets of resources,
3. each resource belonging to one rersource set can communicate via a direct dedicated link with every resource in the other set, and
4. the transfers may occur in any order,

is there a preemptive schedule for performing the transfers whose total length is at most some given bound?

When the problem is stated as an optimization problem the objective is to minimize the schedule length. We call this problem the Simple Data Transfer Scheduling (*SimpleDTS*) problem. An example of this problem was given in

Chapter 1. For the parallel I/O application, it is applicable to systems such as the Sequent [100] where I/O devices are connected to processors via a single shared bus; when discussing scheduling for this application, we sometimes refer to this problem as *Simple I/O Scheduling (SimpleIOS)*. For the communications application, it is applicable to time-slot assignment in TDMA switches which connect a number of input ports to output ports, particularly in the case of satellite switching, where it is of considerable practical interest [75, 74, 9, 10]. (The problem also continues to attract attention in other variations, for example, the multicast version studied by Chen et al. [22], and references therein).

The formal specification of *SimpleDTS* consists of a precedence graph with no edges, a complete bipartite[1] architecture graph, and a bipartite resource graph representing the transfers [76].

In this chapter we also consider an extension to *SimpleDTS* that is useful for modeling some practical parallel data transfer architectures. We call this problem *DTS*, and it is identical to *SimpleDTS* except that the system architecture imposes an additional constraint: at most a fixed number, k, of data transfers may take place at any given time.

For the parallel I/O application, *DTS* arises in multiple-bus systems such as the IBM RP3, where k parallel buses connect processor and I/O devices

[1] A *bipartite* graph is one where the set of vertices can be partitioned into two subsets, i.e., divided into two disjoint exhaustive subsets, such that no edge connects vertices in the same subset. A *complete* bipartite graph is one where there exists an edge between every two vertices that belong to different subsets.

[115]. In general, such multiple parallel bus architectures are attractive for future high-performance parallel computers as they not only allow more than one data transfer to be in progress at any given time, but also allow more processors and devices to be interconnected and improve the system's fault tolerance [107]. For the communications application, *DTS* arises very commonly for all switches where the switching capacity is less than the number of ports, and is useful when the average traffic is much less than the maximum possible traffic.

The applicability of *DTS* to the problem of obtaining optimal time-slot assignment in a TDMA satellite switch allows us to utilize the algorithm **KT** for the satellite switching problem, due to Bongiovanni et al [10], as a starting point for developing an optimal algorithm for *DTS*. By a series of improvements we obtain a faster algorithm for solving *DTS*, improving the time complexity from $O(n^5)$ for Bongiovanni et al's **KT** algorithm, n is the number of resources. Our algorithms are based on optimal k-colorings of bipartite graphs.

This research is both theoretical and experimental. Earlier work by Somalwar [132] on parallel I/O scheduling developed and evaluated heuristics for scheduling of simultaneous requests for multiple resources, such as I/O requests, while Kandappan [85] and Balan [4] studied the impact of data allocation to disks. This chapter formulates the specific parallel data transfer scheduling (or simultaneous multiple resource scheduling) problem discussed above and presents a set of efficient algorithms for this problem, which are then evaluated experimentally for a large range of operating parameters.

3.1 Overview

In section 3.2 we introduce basic definitions and results from graph theory used throughout the chapter. In section 3.3 we present Bongiovanni et al's algorithm in a graph-theoretic form so that it can be applied readily to the DTS problem. We then show how this algorithm can be improved, in three ways. The first improvement arises from the observation that it is not necessary to obtain a min-max bipartite graph matching, as was done in [10], but that a maximum cardinality matching suffices. The first improvement is discussed in section 3.4. The second improvement arises from the observation that a divide-and-conquer strategy can be used to reduce the worst-case time complexity, and is presented in section 3.5. The third improvement arises from observing that the graph of interest can be embedded inside a larger graph, allowing the weighted bipartite matching algorithm of Gabow and Kariv [51] to be applied, hence reducing the time complexity further. This improvement is discussed in section 3.6. In section 3.7 we discuss an experimental study of the efficiency of the scheduling algorithms described in this chapter. The experimental results give rise to some theoretical issues, which are discussed in section 3.8. Finally, in section 3.9 we discuss previous work, and we end with a discussion and suggestions for future work.

3.2 Definitions and problem formulation

The **KT** algorithm [10] was stated in the context of a specific application and developed using a matrix formulation. In this section we introduce the definitions and ideas to be used in section 3.3 to give a graph-theoretic presentation

of the **KT** algorithm, thus allowing it to be applied directly to *DTS*. The key observation is that computing a schedule corresponds to edge-coloring a bipartite graph where the two vertex partitions represent two disjoint sets of resources (say, processors and I/O devices), and edges represent data transfers between them. We first introduce some definitions and notation.

Def. An *edge coloring* of a graph $G = (V, E)$ is a function $c : E \rightarrow N$ which associates a color with each edge such that no two edges of the same color have a common vertex.

Consider two disjoint sets of vertices representing two sets of resources, each of which can participate in at most one data transfer at any given time. Then an edge coloring for a graph G, where each edge of G represents a data transfer requiring one time unit, corresponds to a schedule for the data transfers, and vice versa. To see this, note that all edges of G colored with the same color are independent in that they have no common vertex. Hence the data transfers they represent can be performed simultaneously. An edge coloring of G represents a schedule, where all edges e with $c(e) = i$, for some i, represent data transfers that take place at time i, and vice versa. The number of colors required to edge-color G equals the length of the schedule, and vice versa.

As an aside, we note that edge coloring should not be confused with the classical graph theory problem of vertex coloring, in which vertices are assigned colors such that no two vertices of the same color share an edge. In graph theory terminology, the minimum number of colors required to vertex-color a

graph is called its chromatic number, while we will be interested in the minimum number of colors required to edge-color it, called its chromatic index. In the rest of this thesis, unless explicitly stated, "coloring" a graph refers to edge coloring. For more information on edge coloring, see [41, 5].

Def. A *multigraph* is a graph in which an edge can occur more than once. A *weighted graph* is a graph in which the edges have been assigned weights drawn from the set of natural numbers.

Notation. Let $G = (A, B, E)$ denote a bipartite graph where A and B are two disjoint sets of vertices and $E \subseteq A \times B$ is the set of edges. Let $|A|+|B| = n$ and $|E| = m$. Each edge $e \in E$ has a *weight* $wt(e)$ associated with it, and the weight of a vertex is the sum of the weights of the edges incident upon it. Thus $wt : E \cup A \cup B \rightarrow N$. We can also represent the weighted bipartite graph G as a multigraph $G' = (A, B, E')$ where each edge $e \in E$ is replaced by $wt(e)$ parallel edges of unit weight in E'. Then G is referred to as the *underlying graph* of G', and G' as the multigraph corresponding to G.

Consider an instance of DTS where the architecture allows at most k simultaneous transfers, and data transfers may require an arbitrary positive integer number of time units. Then the problem can be represented as a weighted bipartite graph G, where edge weights represent transfer lengths. Since preemption is allowed, a schedule can be obtained as an edge-coloring of the multigraph corresponding to G, with the restriction that no color may be used more than k times.

We now state the definitions and lemmas used to derive results on edge-

coloring. The following two lemmas are well-known results from graph theory. The rest are graph-theoretic versions of those in Bongiovanni et al [10].

Def. The *degree* of a vertex is the number of edges incident upon it. The degree of a graph is the maximum of the degrees of its vertices. A *critical vertex* is one of maximum degree.

Def. The *weight* of a graph is the sum of the weights of its edges. The *weight* of a vertex is the sum of the weights of the edges incident upon it.

Notice that for a graph with unit-weight edges the degree of a vertex equals its weight.

Def. A *matching* $M \subseteq E$ is a set of edges such that no two edges have a common vertex. A *maximal matching* is one such that no other matching has a larger cardinality. An edge in a matching is said to *cover* the vertices that are its endpoints.

Def. A *critical matching* is one which covers all critical vertices.

Lemma 3.1 *Every bipartite graph has a critical matching.*

proof. See Berge [5]. □

Although the following Lemma is well known, we sketch the proof here as it provides some intuition for results later in the paper.

Lemma 3.2 *Exactly d colors are necessary and sufficient to color a bipartite graph of degree d.*

proof [5]. Clearly, at least d colors are necessary, since a critical vertex requires that each incident edge have a different color. The proof of sufficiency is by induction, sketched as follows. Find a critical matching M, which, from Lemma 3.1, must exist. Color the edges in M a single color and delete them from the graph. The remaining graph has degree $d-1$, and by the induction hypothesis can be colored using $d-1$ colors. Hence the graph can be colored with d colors. □

Def. A *k-coloring* of a graph is an edge-coloring in which each color may be used to color at most k edges.

Lemma 3.3 *At least $q = \max(d, \lceil m/k \rceil)$ colors are necessary to k-color a bipartite graph with m edges, degree d, and unit-weight edges.*

proof. If $d < \lceil m/k \rceil$, at least $\lceil m/k \rceil$ colors are required to color the graph. Otherwise the argument of Lemma 3.2 applies. □

Notation. Let w denote the maximum weight of any vertex in the bipartite graph G, and W denote the weight of G.

Def. The *bound* of the bipartite graph G for an instance of DTS is defined as $q = \max(w, \lceil W/k \rceil)$.

Any scheduling algorithm which produces a schedule of length equal to the *bound* for every instance of DTS, is optimal. Notice that a scheduling algorithm has two measures of performance: optimality, i.e., how close the length of the schedule it produces is to the minimum length, and how long the algorithm takes to run.

Def. For a bipartite graph G representing the transfers in DTS, a *critical weight vertex* is one of weight equal to the bound q. A *critical weight matching* is one which includes all critical weight vertices. A *critical k-matching* is a critical weight matching with k edges.

3.3 An algorithm based on max-min matching (KT)

In this section we present the outline of the algorithm **KT** [10], and in the following section, our first direct improvement to its time complexity. The **KT** algorithm is presented in suficient detail so as to justify the improvement and to provide a basis for the divide-and-conquer algorithm to be presented in section 3.5.

We develop the **KT** algorithm in a graph-theoretic framework, unlike the matrix formulation in [10]. The key observation in Bongiovanni et al [10] is that a bipartite graph can always be augmented to have a certain characteristic, called *k-completeness*, such that a critical k-matching exists. Further, deleting a critical k-matching from the graph leaves it k-complete. Thus a sequence of critical k-matchings can be found, and hence a schedule. The

following two lemmas are used in the proof of Theorem 3.1, which implies that a critical k-matching exists in G.

Def. A graph of weight W and maximum vertex weight w is *k-complete with respect to a constant r* if $w \leq r$ and $W = kr$.

Lemma 3.4 *Any bipartite graph G with bound q can be augmented by adding appropriate weighted edges so as to obtain a bipartite graph H that is k-complete with respect to q, for any $k \leq n$, and further, this can be done in time $O(n)$.*

proof. The proof is constructive, using the algorithm given below.

Algorithm k-complete(G, W, k, q)

Input: a bipartite graph G, eventually to be k-colored, with weight W and bound q.

Output: G with additional edges to make it k-complete with respect to q.

```
1. i, j := 0, 0;
2. while W  < kq
                /* wt(v) is weight of vertex v */
3.      if wt(a(i))  < q and wt(b(j))  < q
4.          add edge (a(i), b(j)) of weight w' = w - max(wt(a(i)), wt(b(j)))
5.          wt(a(i)), wt(b(j)), W += w'
6.      if wt(a(i)) = q
```

```
7.          i := i + 1
8.      if wt(b(j)) = q
9.          j := j + 1
10. end
```

It is quite easy to show that the algorithm is correct [10]. To evaluate its time complexity, observe that at each iteration, either i or j is incremented, or both, and that the procedure will terminate by the time that either i or j equals n. Thus there are at most $2n - 1 = O(n)$ iterations. If the graph is represented as an adjacency list, adding an edge takes time $O(1)$, leading to an overall time complexity of $O(n)$. □

The added weighted edges are called *dummy traffic* and are deleted at the end of the scheduling algorithm.

Def. A bipartite graph is *regular* if all its vertices have the same degree; it is *regular weight* if all vertices have the same weight.

Lemma 3.5 *For any regular weight bipartite graph $G = (A, B, E)$ with $|A| = |B|$, there exists a matching of size $|A|$, which is therefore maximal.*

proof. See Berge [5]. □

Def. A bipartite graph of weight W and maximum vertex weight w is *k-filled with respect to a constant r* if it is k-complete with respect to r, and all vertices have weight r.

A bipartite graph $H = (A, B, E)$ with $|A| = |B|$ that is k-complete with respect to r can be transformed to one that is k-filled with respect to r by a construction sketched as follows. Let $|A| = |B| = n$. Augment H so that $H' = (A \cup C, B \cup D, E \cup F)$ with vertex sets $|C| = |D| = n - k$ and edge set $F = AD \cup BC$ where $AD \subseteq A \times D$ and $BC \subseteq B \times C$. The edges in F are added and their weights assigned such that all vertices in H' have weight r and yet H' is also k-complete with respect to r. We call this algorithm **k-fill**. See Bongiovanni et al [10] for a detailed description of this algorithm and its proof of correctness. We observe that **k-fill** takes time $O(n)$.

Theorem 3.1 [10]. *For any bipartite graph $H = (A, B, E)$ with $|A| = |B|$ and bound q that is k-complete with respect to q, there exists a critical k-matching.*

proof. The proof is by construction, as outlined in the algorithm below.

Algorithm CKM(H, k, q)
Input: a bipartite graph H, which is k-complete with respect to its bound q
Output: A critical k-matching on H.

1. H' = k-fill(H);
2. Find a matching M on H'
3. return(M $\cap$ E)
4. end

Use **k-fill** to generate $H' = (A \cup C, B \cup D, E \cup F)$ from H. From Lemma 3.5, there exists a matching M of size $2n - k$ for H'. Since H' contains no

edges in $C \times D$. $n - k$ edges of M are required to cover the vertices in C, and similarly for D. This leaves $2n - k - 2(n - k) = k$ edges of M having vertices only in H. Further, these k edges cover every critical weight vertex in H, since such vertices do not need to be augmented by adding edges in F. Thus $M \cap E$ gives the required critical k-matching on H. □

We now cite the theorem that, together with Theorem 3.1, ensures that it is always possible to k-color G.

Def. The *duration* of a matching M of a graph with bound q is $r = \min(r', q - r'')$, where r' is the minimum of the edge weights of M and r'' is the maximum vertex weight among vertices not covered by M.

Theorem 3.2 *Let $H = (A, B, E)$ be a k-complete bipartite graph with $|A| = |B|$ which is k-complete with respect to its bound q. Let M be a critical k-matching on H. Let M' be M with the weight of all edges set to the duration r of M. Then the graph $HM = (A, B, E - M')$ is k-complete with respect to $q' = q - r$.*

proof. See Bongiovanni et al [10]. □

Informally, the duration of a matching is the number of time slots a critical k-matching can be used repeatedly. At each time slot that it is used, the weight of all the edges in the graph that are also in the matching decreases by one, since the remaining length of the transfer represented by that edge

decreases by one. A new critical k-matching must be calculated if either the weight of one of the edges of the matching decreases to zero, or a vertex that was previously not critical weight becomes critical.

Theorems 3.1 and 3.2 together lead to an optimal algorithm called **KT** [10]. In **KT**, dummy traffic is added to augment the bipartite graph and make it k-complete with respect to its bound. The well-known max-min bipartite weighted matching algorithm (see [92]), which we call **MaxMinMatch**, is then invoked to find a critical k-matching. The result of Theorem 3.2 is used to calculate its duration. A sequence of critical k-matchings is found by calling the max-min bipartite algorithm repeatedly. Finally, the dummy traffic is removed from the k-matchings to obtain an optimal schedule.

It should be pointed out that the objective of Bongiovanni et al [10] was to obtain a minimum-length schedule while not paying too high a price in terms of L, the number of times that critical k-matchings have to be calculated. The concern in this paper is to obtain a minimum-length schedule while reducing the total running time of the algorithm. In the following we evaluate the running time of **KT** and suggest an improvement.

Theorem 3.3 *The running time of algorithm* **KT** *to find a minimum-length schedule for an instance of DTS is* $O(n^5)$.

proof. Augmenting a bipartite graph to make it k-complete is $O(n)$ (Lemma 3.4), as is deleting the dummy traffic. Bongiovanni et al [10] show that

$L = O(n^2)$, but the running time of **KT** is not calculated. However, we thus know that the number of times that the max-min bipartite matching algorithm **MaxMinMatch** [92] is called is $O(n^2)$. Since the max-min matching algorithm has a running time of $O(n^3)$, the result follows. □

3.4 An improved scheduling algorithm (A1)

In this section we show how **KT** can be improved. We recall that in order to minimize the running time, an algorithm that simply finds a maximum-cardinality matching in H', the regular weight bipartite graph with $2n - k$ vertices in each partition, suffices. Such an algorithm will find the required critical k-matching in H. We can thus use the maximum cardinality matching algorithm of [71], which we call **MaxMatch**. We call the resulting optimal scheduling algorithm **A1**.

Theorem 3.4 *The running time of algorithm* **A1** *to find a minimum-length schedule for an instance of DTS is* $O(n^{4.5})$.

proof. The maximum cardinality matching algorithm **MaxMatch** of [71] takes time $O(|V|^{.5}|E|) = O(|V|^{2.5})$ for a bipartite graph with $|V|$ vertices and $|E|$ edges. For **A1**, $|V| = 2n - k$. Using the maximum cardinality matching algorithm does not affect the worst-case value of L, or the time complexity of adding and deleting dummy traffic. We can use the reasoning of Theorem 3.3 to obtain the result. □

3.5 A divide-and-conquer scheduling algorithm (A2)

In this section we obtain an optimal algorithm for *DTS* which we call **A2**. The key observation is that the bipartite graph G can be partitioned into two graphs of roughly equal weight which represent two independent sub-problems of roughly half the complexity of the original problem. Then algorithm **A1** can be applied recursively to the subproblems to obtain an optimal schedule.

Def. A *walk* is a sequence of distinct adjacent edges. The first and last vertex of the sequence are called the *ends* of the walk. An *open* walk is one in which the ends are distinct; otherwise the walk is *closed.*

A walk differs from a path in that any vertex may be included more than once (not just the first vertex). This definition is used in the following two definitions, which are based on Cole and Hopcroft [29].

Def. An *Euler partition* of a graph is a partition of the edges into open and closed walks, so that each vertex of odd degree is at the end of exactly one open walk, and each vertex of even degree is at the end of no open walk.

Def. An *Euler split* of a bipartite graph $G = (A, B, E)$ is a pair of bipartite graphs $H = (A, B, F)$ and $H' = (A, B, F')$ where $E = F \cup F'$ and a vertex of degree d in G has degree $\lceil d/2 \rceil$ in one of H, H' and $\lfloor d/2 \rfloor$ in the other.

Every graph has an Euler partition, but only bipartite graphs need have Euler splits [5, 29]. An Euler split can be formed from an Euler partition of

G by placing alternate edges of walks into F and F'; both can be found in time $O(n+m)$ for bipartite graphs and multigraphs [48]. An Euler split of the multigraph G' corresponding to G, using the algorithm of Gabow [48], suffices to divide G into two subgraphs each having roughly half the weight of the original. However, this approach would take time proportional to the maximum edge weight in G. In the following we develop an algorithm which is faster because it avoids converting G into a multigraph.

Def. An *Euler division* of a weighted bipartite graph $G = (A, B, E)$ which is k-complete with respect to its bound q is a pair of bipartite graphs $H = (A, B, F)$ and $H' = (A, B, F')$ with $E = F \cup F'$ where the bounds of H and H' are $r = \lceil q/2 \rceil$ and $r' = \lfloor q/2 \rfloor$ respectively, and H, H' are k-complete with respect to their bounds.

In the following we will show that if q is even, an Euler division always exists. (This result is required for the divide-and-conquer algorithm **A2** we develop later in this section). The proof is constructive, and is based on the Euler partition algorithm of Gabow [48] and the *perfect Euler split* algorithm of Somalwar [132], both of which are designed for bipartite graphs with unit-weight edges. We use the notation $wt(Y, y)$ to refer to the weight of a vertex or an edge y in a weighted bipartite graph Y.

Algorithm ED

/* Euler Division of bipartite graphs with even bounds */

Input: Bipartite graph $G = (A, B, E)$ that is k-complete with respect to its bound q, which is even.

Output: An Euler division of G into bipartite graphs $H = (A, B, F)$, $H' = (A, B, F')$.

```
1. F, F' := {}, {};
2. for each e ∈ E such that wt(G, e) > 1 do
3.     wt(H, e), wt(H', e), wt(G, e) := ⌊ wt(G, e) / 2 ⌋, ⌊ wt(G, e) / 2 ⌋,
           wt(G, e) - 2 * ⌊ wt(G, e) / 2 ⌋;
4.     F, F' := F ∪ {e}, F' ∪ {e};
5. end for
6. G' := G;          /* G' introduced for convenience only */
7. P := EP(G');          /* EP generates an Euler partition [48]*/
8. balance := true;
9. for each walk p ∈ P do
10.    for each edge e ∈ p do
11.        if balance = true then
12.            wt(H, e), F := wt(H, e) + 1, F ∪ {e};
13.        else
14.            wt(H', e), F' := wt(H', e) + 1, F' ∪ {e};
15.        end if
16.        balance := ¬balance;
17.    end for
18. end for
```

Lemma 3.6 *Algorithm* **ED** *generates an Euler division of* G.

proof. The first **for** loop divides all even weight edges from G and converts all odd weight edges to unit weight. Thus G' is a unit-weight bipartite graph, and at line 6 H and H' have equal weights and equal degrees at every vertex. Algorithm **EP** generates an Euler partition for a unit-weight bipartite graph [48]. It is clear that lines 9 - 18 ensure that edges from walks in the Euler partition are assigned to H and H' in such a manner that the weights of H and H' differ by at most 1 at the end of the algorithm.

We now show that H, H' are an Euler division of G. Clearly $E = F \cup F'$ by construction. Let X and X' be, respectively, the weights of H and H', x, x' the weights of their critical vertices, and r, r' their bounds. Since q is even, and $W = kq$, W is even. Also, $W = X + X'$ and $|X - X'| \leq 1$ by construction. Hence $X = X' = kq/2$. Consequently, if we show that $r = r' = q/2$, then H, H' will be k-complete with respect to their bounds and thus be an Euler division of G.

By definition of bound, showing $r = r' = q/2$ requires that we show that x, $x' \leq q/2$. Let $z = wt(G, c)$ be the weight of an arbitrary vertex c in G at the start of the algorithm. Note that at line 1, if z is even, there are an even number of odd-weight edges incident upon c in G, and an odd number otherwise. Consequently, after line 6, $wt(G', c) = d$ is even if z is even, and odd otherwise. In either case, after line 6, $wt(H, c) = wt(H', c) = (z - d)/2$ is even. Next, recall that by definition an Euler partition results in an odd-degree vertex being at the end of exactly one open walk; hence

observe that **EP** followed by lines 8 - 18 results in a vertex of weight d in G' contibuting weight at most $\lceil d/2 \rceil$ in H, H'. Therefore, lines 7 - 18 result in $wt(H,c), wt(H',c) \leq (z-d)/2 + \lceil d/2 \rceil = \lceil z/2 \rceil$. Since G is k-complete with respect to q, $z \leq q$, and since q is even, $\lceil z/2 \rceil \leq q/2$. Hence $wt(H,c), wt(H',c) \leq q/2$. □

Lemma 3.7 ED *takes time* $O(n+m)$.

proof. The first **for** loop takes time $O(m)$. From Gabow [48], **EP** takes time $O(n+m)$. The **for** loop from lines 9 - 18 also takes time $O(m)$. □

We can now state the algorithm **A2** which is based on a divide-and-conquer strategy. If q is odd, a critical k-matching of unit duration is found and deleted from the graph to make q even. If q is even, an Euler division is performed and the algorithm applied to the resulting smaller graphs. In the following, angle brackets delimit a sequence and parallel bars denote sequence concatenation.

Algorithm A2.

Input: Bipartite graph $G = (A, B, E)$ with weight function $wt : E \cup A \cup B \rightarrow N$, and integer k.

Output: A minimum-length k-coloring of G, as a sequence s of (M, b) pairs, where M is a critical k-coloring and b is its duration.

0. s := ⟨ ⟩;

1. w := max(wt(v): v ∈ A ∪ B;

2. W := sum(wt(e): e $\in$ E);
3. q := max(w, $\lceil$ W/k $\rceil$);
4. Add dummy traffic to make G k-complete with respect to q.
5. **A2-color**(G); /* Updates s */
6. Delete dummy traffic from s.
7. **end A2.**

Procedure A2-color(G)

1. **if** q is odd **then**
2. M := **CKM**(G); /* **CKM** will find a critical k-matching */
3. s, E, q := s || $\langle$ (M, 1) $\rangle$, E - M, q - 1;
4. **end if**
5. H, H' = **ED**(G);
6. **A2-color**(H);
7. **A2-color**(H');
8. **end A2-color**

Theorem 3.5 *Algorithm* **A2** *finds a minimum-length schedule for an instance of DTS.*

proof. We need to show that **A2** generates a minimum-length k-coloring of G. Lines 1 - 4 and 6 - 7 of **A2** are similar to **KT**. Procedure **A2-color** is called with G, a bipartite graph k-complete with respect to its bound q, as its argument. Thus, from Theorem 3.1, G contains a critical k-matching, which can be found by the procedure **CKM** outlined in the proof of Theorem 3.1.

If q is odd, a critical k-matching is found in line 2 of **A2-color**. Now the graph G with edge set $E - M$ has an even bound $q-1$, and from a corollary of Theorem 3.2, is also k-complete with respect to its bound. Thus from Lemma 3.6, **ED** generates an Euler division of G into H, H', which are k-complete graphs with respect to their bounds $\lfloor q/2 \rfloor$. **A2-color** is applied to them recursively, and by the induction hypothesis generates two k-colorings of length $\lfloor q/2 \rfloor$. Together with the k-matching generated if q is odd, we obtain a k-coloring in s of length q. □

Theorem 3.6 *Algorithm* **A2** *runs in time* $O(Kmn^{.5}\log n)$, *where* K *is the maximum edge weight of* G.

proof. Procedure **CKM** takes time $O(n^{.5}m)$ using the **MaxMatch** algorithm of Hopcroft and Karp [71]. By Lemma 3.7, each invocation of **ED** takes time $O(n+m)$. As long as $K > 1$, **ED** results in two subgraphs with bound $\lfloor q/2 \rfloor$, each having upto n vertices and m edges. When $K = 1$, **ED** results in two subgraphs with bound $\lfloor q/2 \rfloor$, each having roughly $m/2$ edges. Therefore, for $K > 1$, the time $T(q,m)$ for an invocation of **A2-color** with a graph of bound q and m edges is $T(q,m) = O(n^{.5}m) + 2T(\lfloor q/2 \rfloor, m) = KT(q/K, m) + (K-1)O(n^{.5}m) \leq KT(m,m) + KO(n^{.5}m)$. Now $T(m,m) = 2T(m/2, m/2) + O(n^{.5}m)$, i.e., for some $c, m' \in N$, $T(m,m) \leq 2T(m/2,m/2) + cn^{.5}m$ for all $m > m'$. Hence $T(m,m) \leq 4T(m/4, m/4) + 2cn^{.5}m/2 + cn^{.5}m \leq mT(1,1) + (\log m)cn^{.5}m$, for all $m > m'$, i.e., $T(m,m) = O(n^{.5}m\log n)$. Therefore, the total time $T(q,m) = O(Kmn^{.5}\log n)$. □

Thus **A2** is superior to **KT** for problem classes where K is small relative to

$n^{4.5}/(m \log n)$, i.e., say smaller than n^2. **A2** is superior to **A1** when K is small relative to $n^4/(m \log n)$, say smaller than $n^{1.5}$.

3.6 An algorithm for large transfer lengths (A3)

While **A2** is satisfactory for problems where K increases at a modest rate relative to n, it does not handle problems with large transfer lengths well. In fact, **A2** is not a polynomial-time algorithm, but a pseudo-polynomial time algorithm [56]. This is because the weight of each edge in the input graph must be supplied to the algorithm.[2] In this section we describe how an algorithm for edge coloring of weighted bipartite graphs with $k = n$ [51], can be applied for DTS when $k \leq n$. The key observation is that deleting a maximal matching from a k-filled graph (see section 3.3) leaves a k-filled graph.

Consider a weighted bipartite graph $G = (A, B, E)$ with weight W, maximum vertex weight w, maximum edge weight K, and bound q. By Lemma 3.4, it can be converted to a k-complete graph with respect to q by adding dummy traffic, and then to a k-filled graph G' with respect to q using the **k-fill** algorithm This conversion is performed in **KT** and **A1**, and takes time $O(m+n)$.

[2]It takes $O(\log K)$ bits to encode the weight of each edge, so that $O(m \log K)$ bits are needed to supply the information about the edge weights in the input graph. On the other hand, it takes $O(\log n)$ bits to encode each vertex label, and hence $O((m+n)\log n)$ bits to encode the graph connectivity. Thus the total length of the string describing the input is $I = O(n^2(\log n + \log K))$. The time complexity of **A2** increases as a polynomial in I if only n increases, but as an exponential in I if K increases.

Let $G' = (A', B', E')$ with $|A| = |B| = 2n - k$, weight $W' = (2n - k)q$, maximum edge weight K', and all vertex weights equal to q. In both **KT** and **A1**, a maximal matching M' of size $2n - k$ on G' is obtained, and the k edges of $M = M' \cap E$ are deleted from G. This results in a new graph H which is k-complete with respect to $q - 1$ but not k-filled with respect to $q - 1$. It is necessary to invoke **k-fill** on H before obtaining the next critical k-matching. We observe that if all $2n - k$ edges of M' are deleted from G', the result is a graph G'' that is k-filled with respect to $q - 1$, obviating the need for k-filling again before obtaining the next matching.

The argument above leads to the following algorithm, where an edge-coloring on G' is used to obtain a series of maximal matchings on G', which are then pruned to obtain a series of critical k-matchings on G. The **weighted-edge-coloring** algorithm of Gabow and Kariv [51] is used to edge-color G'. It should be pointed out that the space complexity of the **weighted-edge-coloring** algorithm, and hence of **A3**, is high: $O(mn \log K)$.

Algorithm A3.

Input: Bipartite graph $G = (A, B, E)$ with bound q and maximum edge weight K.

Output: A minimum-length k-coloring of G, as a sequence s of (M, b) pairs, where M is a critical k-coloring and b is its duration.

0. s := $\langle\ \rangle$;
1. Add dummy traffic to make G k-complete with respect to q.
2. G' := k-fill(G);

3. C := **weighted-edge-color**(G'); /* C is an edge coloring of G'*/

4. **for** each color $c \in C$

5. M', b := edges colored by c, duration c is used; /* M' is max. matching on G' */

6. M := M' $\cap$ E;

7. s := s || $\langle$ (M, b) $\rangle$;

8. **end for**

9. Delete dummy traffic from s.

10. **end A3.**

Theorem 3.7 *Algorithm* **A3** *takes time* $O(n^3(\log n + \log K))$.

proof. As noted earlier, k-**fill** takes time $O(n)$. The weighted edge coloring algorithm [51] takes time $O(|V||E| \log J)$ for a weighted bipartite graph with largest edge weight J. In **A3** we apply this algorithm to G', where $|V| = 2n-k$, $|E| \leq (2n-k)^2 = O(n^2)$, and $J = K' \leq q \leq mK$. Thus the total time is $O(n^3(\log n + \log K))$ to obtain an edge-coloring of G'. Since the coloring algorithm [51] uses $O(|E| \log J)$ colors, the number of iterations of the **for** loop is $O(n^2(\log n + \log K))$. Since $|M'| = 2n - k$, lines 5 - 7 can be implemented in time $O(n)$, and thus the **for** loop takes time $O(n^3(\log n + \log K))$. □

Algorithm **A3** is faster than **KT** and **A1** for a large class of graphs, i.e., when $\log K$ is small relative to $n^{1.5}$. Note that **A2** is still faster than **A3** for $K = 1$ or when K is bounded by a small constant.

3.7 Experimental evaluation

In this section we present the results of an experimental evaluation of the algorithms described in this chapter. This work has confirmed that our algorithms provide results that are superior to the **KT** algorithm for the situations studied. This experimental work extends the previous related work of Somalwar [132], Kandappan [85], Balan [4] and Jain et al [78]. It has also led to the investigation of even faster algorithms for data transfer scheduling, which are heuristic in nature [76], and are discussed in a subsequent chapter.

We compare the effects of using the four scheduling algorithms discussed in this chapter, namely **KT**, **A1**, **A2**, and **A3**. Since all four algorithms produce optimal schedules, the key question is the amount of time taken to produce those schedules. There are four parameters that affect the performance of these algorithms: the number of vertices in each partition of the graph (n_1 and n_2), the number of edges m, the maximum number of simultaneous transfers allowed k, and the maximum edge weight K. When the number of vertices in each partition is the same, we set $n = n_1 = n_2$. We evaluate the bahavior of the algorithms as each of these parameters is varied.

Scenario: Volume Visualization of Scientific Data. Consider a scenario where users, who may be physicians, health care workers, scientists, etc., need to share and access a large image database. The images may consist of medical information, e.g. computer-aided tomography (CAT) scans, or oil prospecting information, e.g. seismic data from acoustical depth soundings, and so on. The database is processed and stored at a parallel computer

site, and users view the images by requesting image files to be displayed on their graphics workstations. The parallel computer is a shared-bus system, in which processors and disks are connected to a set of common buses (or a single high-speed system bus that is shared in a time-multiplexed fashion), which allow multiple I/O transfers to proceed in parallel. In order to provide a reasonable response time for the users, the workstations are also connected to the common buses. A user request for an image file is processed by the CPUs at the parallel computer, and results in image data being transferred from the system disks to the user's workstation across the common buses. The workstations off-load some low-level image-processing tasks from the parallel computer, such as rendering, shading, etc. In this scenario, the parallel computer's operating system batches the image file requests and schedules the resulting I/O transfers.

In terms of this parallel I/O application, n_1 and n_2 correspond to the number of disks and workstations, m the number of image files to be transferred, k the number of parallel buses in the system including the degree to which they can be time-multiplexed, and K the longest file length in disk blocks. Using the parallel I/O application as a context helps to bound the ranges of values of the parameters for which the scheduling algorithms are evaluated. Given the scalability problems of shared-memory shared-bus parallel computer systems, we evaluate the algorithms for relatively modest numbers of disks and workstations ($n_1, n_2 < 256$). Typically, we choose $n = n_1 = n_2 = 64$. Consistent with this context, we also assume only a relatively modest number of simultaneous parallel transfers ($4 \leq k \leq 16$, and $k < n_1, n_2$). As far as the number of transfers is concerned, we choose $100 \leq m \leq 1000$ as a reasonable range considering the image database scenario sketched above. The maximum

file length, K, is increased systematically until the behavior of the program seems to become clear from the trend of the increase in CPU time.

The algorithms were implemented as programs in C, generally following the outlines sketched in this chapter. The implementation of **KT** and **A2** was a non-trivial adaptation from the implementation of Somalwar [132], which handles unit-weight edges only. The Hopcroft and Karp [71] maximum cardinality matching algorithm used in **A1** and **A2** was implemented in a form very similar to that given in the text by Moret and Shapiro [106]. In the implementation of **A2** we made two modifications: the recursive structure of the program was replaced by iteration, and rather than performing k-filling at every iteration, it was performed only at the start of the algorithm.

The main implementation difficulty was with the **weighted-edge-color** algorithm of Gabow and Kariv [51], especially since their description omits two important points. The first is regarding the conditions under which new colors are assigned to edges; it was necessary to carry out a detailed case analysis of the situations in which the weighted augmenting path algorithm should in fact assign new colors to uncolored edges, taking into account the special cases that occur when all the remaining uncolored edges are of unit weight. The second omission was more serious, and was the key observation that as one weighted colored edge is partially assigned a new color, all edges in the graph bearing the old color must also be partially assigned the new color. Again, special cases arise when all uncolored edges are unit-weight. These two points are very important as they form the basis for guaranteeing that the number of colors used in **weighted-edge-color**, and hence its time complexity, is

logarithmic in K rather than linear. Implementing **weighted-edge-color** to correctly handle these two points is quite involved.

The programs implementing **KT** and **A1-A3** were evaluated by measuring the CPU time they take to execute when presented with uniformly randomly generated bipartite graphs as inputs. Random graphs were generated for selected combinations of the n_1, n_2, m and K program parameters using a pseudo-random number generator [94]. The programs were executed on a Sun Sparc 2 workstation running the SunOS^{TM} Release 4.1.1 operating system, after being compiled using the Sun Microsystems C compiler (bundled with SunOS Release 4.1.1), with Level 4 optimization enabled ("-O4" option). The data structures for the programs all fit in the 32 MB main memory of the workstation, and so the programs do not perform any I/O in order to execute, except to read the input graph and print results.

The CPU time taken by a program for each input random graph was measured by the C Shell "time" command. Although this measurement tool has a resolution of only 20 ms, it was not thought necessary to use a higher resolution measurment (e.g. the system's real-time clock) since most measurements we take are on the order of seconds and the programs tend to display rather pronounced differences in their execution times. For each selected combination of the program parameters, one hundred random graphs were generated, and the CPU time taken by each program, as reported by the "time" command, was recorded for each input graph. The mean and standard deviation of each set of 100 measurements was calculated using the programs given in *Numerical Recipes* [119]. The data was plotted using the DeltaGraph Professional^{TM} software package, on a Macintosh system. The

same software was also used to generate smooth curve fits based on models that were supplied to the program as candidates.

In the following we present the results of the experiments, the calculated means, and plotted points and curve fits for each progream as each of the four parameters, m, m, n and K were varied. Although the data presented here are for $n = n_1 = n_2$, in a previous study we have considered the case $n_1 \neq n_2$ and found qualitatively similar results [78].

3.7.1 Effect of varying the number of transfers

In Fig. 3.1 we plot the mean CPU time taken by each program implementing **KT**, **A1**, **A2** and **A3** for inputs where the parameters $n = n_1 = n_2 = 64$, $k = 4$ and $K = 1$ are fixed and m varies from 100 to 1000. That is, we see the effect of varying the number of transfers while keeping all other parameters fixed. Each data point represents the mean of 100 measurements of CPU time, and the error bars indicate one standard deviation above and below the mean. The curve-fits shown in Fig. 3.1 correspond to the following equations and correlation coefficients:

$$KT(t) = 1.55 \times 10^{-6}\, m^2 + 2.18 \times 10^{-3}\, m - 2.18 \times 10^{-2}\ ,$$

$$R2^2 = .98,\ R1^2 = .99,\ R0^2 = .99$$

$$A1(t) = 1.73 \times 10^{-6}\, m^2 + 1.48 \times 10^{-3}\, m - 6.66 \times 10^{-2}\ ,$$

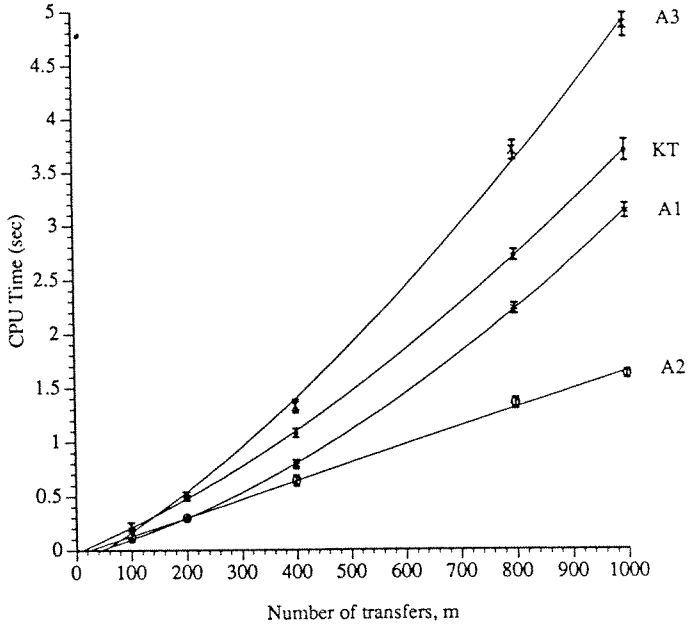

Figure 3.1: CPU time versus number of transfers for $n = 64$, $k = 4$, $K = 1$

$$R2^2 = .99,\ R1^2 = .98,\ R0^2 = .99$$

$$A2(t) = 1.69 \times 10^{-3}\, m - 3.82 \times 10^{-2}\,, \qquad R^2 = .99$$

$$A3(t) = 7.37 \times 10^{-4}\, m \log m - 0.258, \qquad R^2 = .99$$

The results plotted and curve-fitted in Fig. 3.1 display some interesting characteristics. It is interesting to see that the execution time of **A2** in these experiments increases very close to linearly with m, exactly as predicted by the theoretical worst-case time complexity formula $O(Kmn^{.5} \log n)$ derived for **A2** in the previous section. On the other hand, we see that the other three algorithms also show marked increases with m, which are not predicted by the theoretical complexity analysis. This discrepancy is discussed in the next section of this chapter.

Qualitatively, however, we see that for this combination of parameters, **A2** out-performs the other algorithms significantly, and is likely to continue doing so as m increases.

3.7.2 Effect of varying the degree of data transfer parallelim

In Fig. 3.2 the parameters $n = n_1 = n_2 = 64$, $m = 1000$, and $K = 1$ are fixed and k varies. That is, we see the effect of varying the degree of parallelism in the data transfer while keeping all other parameters fixed. The equations corresponding to the curve fits are given by:

$$KT(t) = 15.33\ k^{-1.02}, \qquad R^2 = .99$$

$$A1(t) = 12.23\ k^{-0.98}, \qquad R^2 = .99$$

$$A2(t) = 5.13\ k^{-0.84}, \qquad R^2 = .99$$

$$A3(t) = 30.49\ k^{-1.33}, \qquad R^2 = .99$$

For all four algorithms, the CPU time varies inversely with k, a trend not predicted by the theoretical time complexity formulas derived earlier. It is interesting to see that for all practical purposes the variation is proportional to $1/k$ for **KT** and **A1**. This is discussed in the next section.

Qualitatively, we again observe that for this set of parameters, **A2** out-performs the other algorithms signficantly.

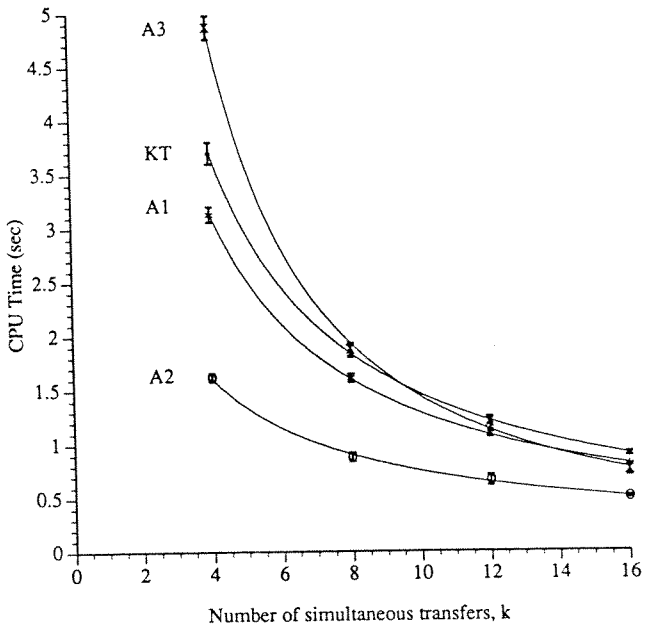

Figure 3.2: CPU time versus number of simultaneous transfers for $n = 64$, $m = 1000$, $K = 1$

3.7.3 Effect of varying the number of resources

In Fig. 3.3 the parameters $k = 4$, $m = 1000$, and $K = 1$ are fixed and $n = n_1 = n_2$ is varied. That is, we see the effect of varying the number of resources in the system while keeping all other parameters fixed. The curve fits are given by:

$$KT(t) = 4.45 \times 10^{-4}\, n^2 + 1.01 \times 10^{-2}\, n + 1.26,$$

$$R2^2 = .99,\ R1^2 = .96,\ R0^2 = .99$$

$$A1(t) = 1.17 \times 10^{-3}\, n^{1.5} + 1.91 \times 10^{-1}\, n^{0.5} + 1.26, \qquad R^2 = .99$$

$$A2(t) = 1.18 \times 10^{-2}\, n \log n + 1.55, \qquad R^2 = .99$$

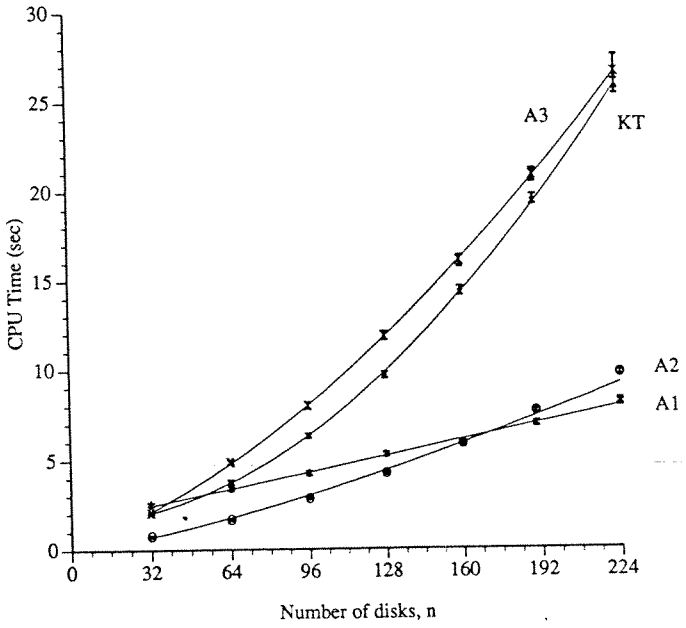

Figure 3.3: CPU time versus number of resources for $k = 4$, $m = 1000$, $K = 1$

$$A3(t) = 2.75 \times 10^{-4}\, n^2 + 5.68 \times 10^{-2}\, n + 0.02,$$

$$R2^2 = .99,\ R1^2 = .98,\ R0^2 = .99$$

We observe that, at least for the set of experiments described here, the performance of both **KT** and **A1** appears to be significantly better than that predicted by their theoretical complexity formulas $O(n^5)$ and $O(n^{4.5})$ respectively. This is discussed in the next section.

Qualitatively, we observe that for this set of parameters **A1** and **A2** significantly out-perform the other algorithms, with **A1** likely to have better performance than **A2** only for $n > 160$.

3.7.4 Effect of large transfer lengths

In Fig. 3.4 the parameters $n = 64$, $k = 4$, and $m = 1000$, are fixed while K is varied. Thus the same number of transfers have to take place for all runs, but their lengths are integers drawn uniformly at random from the interval [1, K]. The curve fits are given by:

$$KT(t) = 0.23 \log K + 0.25, \qquad R^2 = .96$$

$$A1(t) = 0.14 \log K + 0.13, \qquad R^2 = .95$$

$$A2(t) = 0.08K + 0.09, \qquad R^2 = .99$$

$$A3(t) = -8.65 \times 10^{-4} K^2 + 0.18K + 0.29,$$

$$R2^2 = .82, \; R1^2 = .96, \; R0^2 = .99$$

We observe a number of interesting features in this set of curves. The first is that although the CPU time behavior of both **KT** and **A1** can be fit to an $O(\log K)$ curve, the constants involved are so small that it is essentially independent of K for $K > 10$. This is as predicted by the theoretical time complexity analysis. We also see that the CPU time for **A2** increases very close to linearly with K, again as predicted by theoretical analysis. On the other hand, the behavior of **A3** does not follow $O(\log K)$ for this range. This is discussed in the following section.

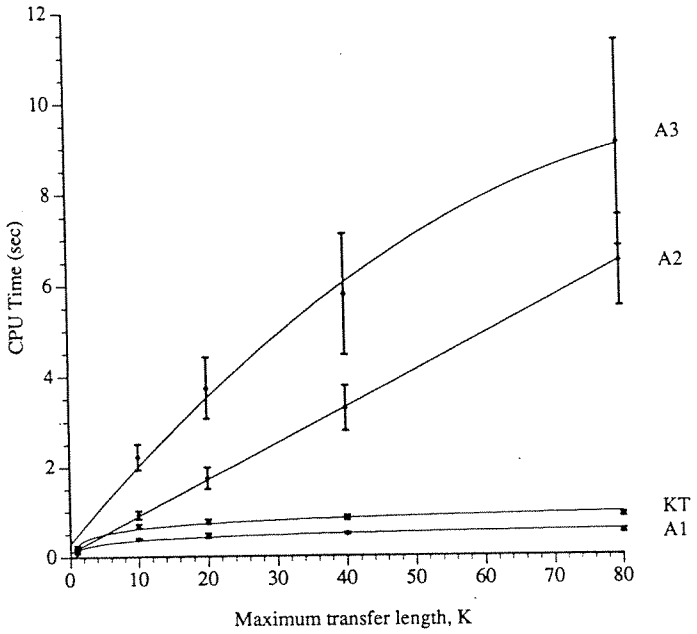

Figure 3.4: CPU time versus maximum transfer length $n = 64$, $k = 4$, $m = 1000$

Qualitatively, we observe that **KT** and **A1** significantly out-perform the other algorithms, both in terms of absolute CPU time and in terms of its variance for random input instances. Between the better two algorithms, **A1** consistently out-performs **KT**.

3.8 Interaction of theoretical and experimental evaluation

Our study of the four scheduling algorithms **KT**, and **A1** - **A3**, is an interesting example of the importance of cross-checking theoretical and experimental evaluations of algorithm behavior. In several sets of experiments described above, we found that the measured time behavior of the algorithm differs significantly from that predicted by theoretical analysis alone. These discrep-

Algorithm	Method	m	k	n	K
KT	theory	const.	const.	n^5	const.
	experiment	m^2	$1/k$	n^2	const.
A1	theory	const.	const.	$n^{4.5}$	const.
	experiment	m^2	$1/k$	$n^{.5} + n^{1.5}$	const.
A2	theory	m	const.	$n^{.5} \log n$	K
	experiment	m	$1/k^{.84}$	$n^{.5} \log n$	K
A3	theory	const.	const.	$n^3 \log n$	$\log K$
	experiment	$m \log m$	$1/k^{1.3}$	n^2	K^2

Table 3.1: Asymptotic theoretical vs. experimental behavior of algorithms as input parameters vary. (See following Table also)

ncies motivate us to re-examine our theoretical and experimental evaluation more closely, leading to a better understanding of the algorithms' behavior.

The discrepancies we observe are summarized in Table 3.1, where we show the asymptotic theoretical and experimental behavior of the algorithms as each of the parameters m, k, n and K is varied. (The curve-fitted constants have been omitted from the experimental results as we are only considering the estimated asymptotic algorithm behavior). In this section we show that most of these apparent discrepancies can in fact be resolved by one of two ways: more sophisticated theoretical analysis, and further experimentation.

3.8.1 Effect of number of transfers

We first consider the marked discrepancy between theoretical and experimental behavior of **KT** and **A1** as m is varied. We re-examine the theoretical time complexity analysis presented earlier in this chapter and observe that it can be made more precise in two ways.

Observation. The time complexity of the max-min bipartite weighted matching algorithm [92], **MaxMinMatch** can be refined to $O(mn)$ from $O(n^3)$. Similarly, the time complexity of the maximum cardinality matching algorithm **MaxMatch** [71] can be refined to $O(mn^{.5})$ from $O(n^{2.5})$. □

Lemma 3.8 *The number of times that critical k-matchings have to be calculated in either* **KT** *or* **A1**, *i.e., the number of iterations L, can be refined to* $L = O(m+n)$ *from* $L = O(n^2)$.

proof. Recall that a new critical k-matching must be calculated, in both **KT** or **A1**, if either the weight of one of the edges covered by the matching decreases to zero, or a vertex that was previously not critical weight becomes critical. Thus at every iteration of the algorithm, either an edge can be deleted, or a vertex becomes critical, or both. Now observe that once a vertex becomes critical it remains critical until the algorithm terminates (since it must continue to receive full service if the algorithm is to produce a schedule which meets the lower bound on schedule length). Hence any vertex can be promoted from being non-critical to critical at most once during the execution of the algorithm. Also, any edge can be deleted at most once. Since at each iteration at least one of these events (edge deletion or vertex promotion) occurs, there are at most $O(m+n)$ iterations. □

A careful reading of the proof in [9] shows that the authors have used a similar reasoning to the one we have presented above, but have set $m = O(n^2)$, leading to the $L = O(n^2)$ bound.

Theorem 3.8 *The time complexity of algorithm* **KT** *can be refined to* $O(m^2n + mn^2)$ *from* $O(n^5)$, *and of algorithm* **A1** *to* $O(m^2n^{.5} + mn^{1.5})$.

proof. The time complexity of both **KT** and **A1** is given by L times C, the time for the critical k-matching algorithm used. From the observation, $C = O(mn)$ for **KT** and $C = O(mn^{.5})$ for **A1**. From Lemma 3.8, $L = O(m+n)$. The theorem follows. □

We can use this result to explain the experimentally observed variation of **KT** and **A1**'s running time not only with m, but with n.

We now consider the discrepancy between the theoretical time complexity of **A3**. $O(n^3(\log n + \log K))$, and the experimentally observed variation with m. Recall that **A3** performs k-**fill** on the input graph G with n vertices and m edges to produce a graph G' which is then colored using the **weighted-ege-color** algorithm of Gabow and Kariv [51]. We first show that k-filling G only increases the number of edges in G' to $O(m+n)$ instead of $O(n^2)$.

Lemma 3.9 *A bipartite graph with* n *vertices and* m *edges can be converted to a* k*-filled graph with at most* $O(m+n)$ *edges, assuming that weighted edges can be used for* k*-filling.*

proof. Let $G = (A, B, E)$ be the input graph, w its vertex weight and W its total weight. For ease of exposition we assume $|A| = |B| = n$ in the following; the case $|A| \neq |B|$ is very similar and left to the reader.

From the time complexity analysis of the **k-complete** algorithm we see that at most $O(n)$ edges are added during this phase. The k-filled bipartite graph $G' = (A \cup C, B \cup D, E \cup E')$ is obtained by adding $n - k$ vertices to each partition of G and by adding edges $E' \subseteq (A \times D) \cup (B \times C)$ until all vertices have weight q.

It has been shown that k-filling can always be done [10]. We show by induction the proposition that k-filling G requires adding at most $4n$ edges, as follows. It suffices to consider vertices in only one partition of G, say A, with the number of its vertices examined, j, being the induction variable. Initially the first vertex $a_1 \in A$ has been examined. An edge of weight $q - wt(a_1)$ is added from a_1 to d_1, the first vertex in D; the proposition holds. Assume that after j vertices have been examined, at most $2j$ edges have been added. When a_{j+1} is examined, let d_k be the first vertex in D with weight less than q. An edge (a_{j+1}, d_k) of weight $w' = q - \max(wt(a_{j+1}, wt(d_k))$, and another edge (a_{j+1}, d_{k+1}) of weight $q - w'$, are all that are needed to make $wt(a_{j+1}) = q$. Thus the induction is complete and the proposition holds.

It follows that if non-unit-weight edges are used for k-filling, G' has $O(m+n)$ edges. □

Notice that making a graph **k**-filled with respect to its bound increases not only its number of edges but also the maximum edge weight. In fact, during **k-complete**, for instance, the maximum edge weight can increase from K to as much as mK. To see an example of this, consider the graph with $n = 5$, $m = 4$ edges each of weight $K = 2$ connected from a single vertex $a \in A$

to four distinct vertices in B, and given $k = 3$. Then $w = W = 8$, and $q = 8$, so that edges of total weight $kq - W = 16$ need to be added while not increasing the maximum vertex weight. This can be done by adding three edges, of weights 8, 6 and 2, respectively, making $K' = 8 = mK$. However, the the bound of the graph, q, remains unchanged, and so the time complexity analysis of **A3** and other algorithms is not affected.

Theorem 3.9 *The time complexity of* **A3** *can be refined to* $O((nm + n^2)(\log m + \log K))$ *from* $O(n^3 (\log n + \log K))$.

proof. The **weighted-edge-color** algorithm of [51] takes time $O(|V'|\ |E'|\ \log J)$ where V' is the total number of vertices and J is the maximum edge weight of G'. While $|V'| = O(n)$ and $J \leq mK$, from Lemma 3.9 we have $|E'| = O(m + n)$, giving the new complexity estimate. A similar argument holds for the **while** loop in the **A3** algorithm. □

As an aside, we make the following observation here, which will be used in a later section.

Observation. If only unit-weight edges can be used for k-filling, G' can have upto $O(nm)$ edges. □

3.8.2 Effect of data transfer parallelism

We consider the effect of increasing k, the number of simultaneous transfers posssible, upon the behavior of the four scheduling algorithms.

Informally, we can consider this effect by recalling that the length of the schedule is given by the bound of the graph, $q = \max(w, \lceil W/k \rceil)$. In general, as k decreases the second component of q dominates, and the schedule length increases. Since all four algorithms are essentially iterative computations of matchings (or, in the case of **A3**, of augmenting paths), and the number of iterations increases with the schedule length, as k decreases the execution time of all four algorithms increases. For situations where $\lceil W/k \rceil > w$, the CPU time for all four algorithms should vary as $1/k$.

3.8.3 Effect of transfer lengths

The theoretical advantage of **A3** over **A2** is that its time complexity is polynomial in K rather than a pseudo-polynomial. However, while **A2**'s CPU time increases linearly with K as expected for $1 \leq K \leq 80$, **A3** appears to perform much worse.

We observe from the variation of **A3** with m, n, and k that it appears to have much larger constants of variation than the other three algorithms. We thus extended the investigation of **A3**'s behavior to larger values of K in order to estimate its asymptotic behavior. However, as noted earlier, **A3**'s storage requirements are very high. In order to keep all data structures in memory, experiments for $K > 80$ could not be conducted on the Sun Sparc 2 workstation, which has 32 MB of main memory. The program was run instead on a Solbourne Series5e/900^{TM} workstation with 128 MB of main memory. The workstation runs Solbourne's UNIX-like OS/MP 4.1A.1 and can run programs compiled for the Sun Sparc 2 without recompilation. Input

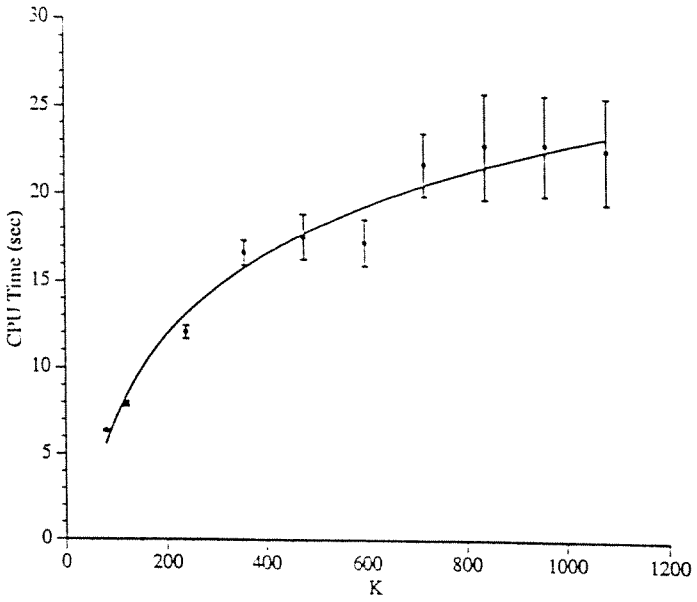

Figure 3.5: CPU time on Solbourne versus maximum transfer length for $n = 64$, $k = 8$, $m = 100$

graphs, time measurements, and mean CPU times were generated as before, and plotted with error bars and curve fits as before, to yield the plot in Fig. 3.5. Notice the relatively large standard deviation of the measurments. The curve fit for $K \geq 80$ is given by:

$$A3(K) = 9.83 \log K - 24.2, \qquad R^2 = 0.96$$

The revised comparison of theoretical and experimental results in Table 3.2 displays good agreement between the two.

Algorithm	Method	m	k	n	K
KT	theory	m^2	$1/k$	n^2	const.
	experiment	m^2	$1/k$	n^2	const.
A1	theory	m^2	$1/k$	$n^{.5} + n^{1.5}$	const.
	experiment	m^2	$1/k$	$n^{.5} + n^{1.5}$	const.
A2	theory	m	$1/k$	$n^{.5} \log n$	K
	experiment	m	$1/k^{.84}$	$n \log n$	K
A3	theory	$m \log m$	$1/k$	n^2	$\log K$
	experiment	$m \log m$	$1/k^{1.3}$	n^2	$\log K$

Table 3.2: Revised asymptotic theoretical vs. experimental behavior of algorithms as input parameters vary

3.9 Discussion

3.9.1 Previous related work

Algorithms **A1** - **A3** provide a method for scheduling parallel data transfers such as I/O in multiple-bus parallel computers and time slots in TDMA switches, and are a substantial improvement in both theoretical and experimentally observed execution time over algorithm **KT**. Since **A1** - **A3** are framed in terms of optimal edge-coloring of bipartite multigraphs, we review this literature briefly. In Table 3.3 we summarize the previous work. We emphasize that almost all the literature surveyed consists of theoretical work only. Very few studies of the type reported in this chapter have been performed to experimentally evaluate the behavior of the scheduling and coloring algorithms that have been developed.

First consider the optimal edge-coloring of bipartite graphs with unit-weight edges. Vizing [137] developed an algorithm using the basic notion of augmenting paths that takes time $O(mn)$. Gabow [48] exploited the partitioning

of a bipartite graph by means of Euler partitions in order to apply the divide-and-conquer strategy to edge coloring algorithms. This idea has been used repeatedly [50, 51, 29, 132, 78]. In the same paper Gabow noted that the Mendelsohn-Dulmage method (see [92]) could be used to construct an algorithm to obtain a matching covering all vertices of maximum degree. This matching algorithm takes time $O((m+n)n^{0.5})$ and was used in conjunction with the divide-and-conquer strategy to obtain an edge coloring algorithm of time complexity $O(n + mn^{0.5} \log n)$.

Gabow [48] also observed that in the special case that the degree of the graph is a power of 2, the divide-and-conquer algorithm need never call the matching algorithm, allowing an edge coloring to be found in time $O(n + m \log n)$. Gabow and Kariv [50, 51] use this observation to obtain faster edge-coloring algorithms for graphs whose degree is not a power of 2, by repeatedly constructing partially-colored subgraphs whose degree is a power of 2 and coloring them efficiently.

Finally, Cole and Hopcroft [29] use the idea of Euler partitioning the graph in order to design a matching algorithm that covers all vertices of maximum degree. This matching algorithm runs in time $O(\max(m, n \log n \log^2 d))$, where d is the graph degree, and is faster than the Mendelsohn-Dulmage matching method used by Gabow [48] by a factor of roughly $O(n^{0.5})$; it leads to an $O(m \log n)$ algorithm for edge coloring.

There has been relatively little attention paid to the problem of edge-coloring weighted bipartite graphs. Before we review this literature, it is interesting

Reference	Unit-Weight Edges *(K = 1)*	Unequal Weight Edges *(K ≥ 1)*
Unlimited transfers, $k = n$:		
Vizing, 1964 [137]	mn	
Gonzalez and Sahni, 1976 [62]	m^2	m^2
Gabow, 1976 [48]	$n + mn^{0.5} \log n$	
Gabow and Kariv, 1978 [50]	$mn^{0.5} \log n$	
Gabow and Kariv, 1982 [51]	$m \log^2 n$; $n^2 \log n$	$nm \log K$
Cole and Hopcroft, 1982 [29]	$m \log n$	
Limited transfers, $k \leq n$:		
Bongiovanni et al, 1981 (**KT**) [9, 10]	$m^2n + mn^2$	$m^2n + mn^2$
Somalwar, 1988 [132]	$mn^{1.5} \log n$	
A1	$m^2n^{0.5} + mn^{1.5}$	$m^2n^{0.5} + mn^{1.5}$
A2	$mn^{0.5} \log n$	$Kmn^{0.5} \log n$
A3	$(n^2 + nm) \log m$	$(n^2 + nm)$ $(\log m + \log K)$

Table 3.3: Summary of previous related work

to discuss an influential early application, that of constructing class-teacher timetables [63], also called the timetabling problem [8, 31]. To quote Gotlieb in 1962, "For a high school with thirty or more classes, even after the sets of teachers to be associated with a given class are assigned, it takes many man-weeks to draw up a schedule specifying when teachers and classes are to meet. ... [I]n the Metropolitan Toronto area alone, over sixty large time-tables are drawn up annually" [63].

The class-teacher timetabling application in its simplest form can be modeled as a problem of edge-coloring a weighted bipartite graph, where the vertex partitions represent teachers and classes, the edges represent the teachers assigned to a class, and edge weights represent the number of contact hours [8]. The application as specified by Gotlieb included the additional prac-

tical constraints that the total number of time slots available is fixed and that for certain time slots either a class or a teacher is unavailable; for these constratints the scheduling problem is NP-complete [40]. However, Gotlieb's application has continued to attract attention because of its practical importance and theoretical applicability [31, 142].

It is interesting to note that possibly the earliest efficient algorithm for edge-coloring weighted bipartite graphs [62] was developed in the context of a scheduling application, namely the scheduling of jobs in an open shop[3] Gonzalez and Sahni's algorithm [62] finds shortest augmenting paths to obtain matchings, a strategy similar to that used in Hopcroft and Karp's matching algorithm [71], and thereby obtain an edge coloring algorithm that runs in time $O(m^2)$. Gabow and Kariv [51] again exploit the observation that graphs whose degree is a power of 2 can be colored efficiently to obtain an algorithm that takes time $O(nm \log K)$, where K is the largest edge weight; this is faster than the algorithm of Gonzalez and Sahni [62] for a large class of graphs, i.e., whenever $n \log K = o(m)$.

The problems we address in this chapter, *SimpleDTS* and *DTS*, are modeled as optimal k-coloring of a bipartite graph, i.e., optimal edge coloring a bipartite graph where each color can be used to color at most k edges. Optimal k-coloring also directly models the class-teacher timetabling application if at most k classrooms are available at any given time [8]. However, to our knowledge, the only previous algorithmic solutions for optimal k-coloring of

[3] An open shop is the job-shop problem (defined in Chapter 2) with the restriction that the precedence graph has no edges, i.e., tasks can be performed in any order.

bipartite graphs are by Bongiovanni et al [10] and Somalwar [132]. Algorithm **KT** of Bongiovanni et al [10] can operate on graphs with non-unit edge weights, and, as described in this chapter, takes time $O(m^2n + mn^2)$. The algorithm of Somalwar [132] is applicable only to graphs with unit edge weights, and takes time $O(n^{1.5}q \log q)$, where q is the bound of the graph, i.e., takes time $O(mn^{1.5} \log n)$. As shown in this chapter, our algorithms **A1 - A3** are faster than either of these algorithms.

It is interesting to compare the performance of the algorithm of Somalwar [132] and **A2** for unit-weight edges (see Table 3.3), since **A2** basically extends Somalwar's algorithm to the unequal weight edges case. The reason that **A2** is faster by a factor of n, even if only unit-weight edges are allowed in the input graph, is that **A2** can handle weighted edges being introduced by k-**fill**, while Somalwar's algorithm cannot. Thus for **A2** the k-filled graph has only $O(m+n)$ edges instead of $O(mn)$ edges (see Lemma 3.9 and the Observation following it). The increase in the number of edges that must be processed by Somalwar's algorithm accounts for its time complexity being higher by $O(n)$.

It is for this reason also that any simplistic application of the k-filling technique to a previous unit-weight edge coloring algorithm will result in an algorithm that performs worse than **A2**. For example, consider a simple extension of Cole and Hopcroft's [29] algorithm to handle $K > 1$ and $k \leq n$. Firstly, weighted edges will be replaced by unit-weight edges, making the number of edges $O(mK)$. Then the k-filling technique will increase the number of edges to $O(mnK)$, yielding an $O(Kmn \log n)$ algorithm for edge coloring, which is worse that **A2** by a factor of $O(n^{0.5})$. (As an aside, we observe that Cole

and Hopcroft's algorithm can be extended in this way to perform better than Somalwar's).

Similarly, a simple extension of Gonzalez and Sahni's [62] algorithm to handle $k \leq n$ by k-filling will result in an algorithm that is slower than **A2** for bipartite graphs with unit-weight edges by a factor upto $O(n)$; for weighted edges it will be faster than **A3** by a factor $O(\log n + \log K)$ for sparse graphs, and slower by a factor $O(n/(\log n + \log K))$ for dense graphs. (However, such an extension to Gonzalez and Sahni's algorithm may be useful in practice, as it is relatively simple to implement and takes much less space than **A3**).

In summary, algorithms **A1** - **A3** generalize previous edge coloring algorithms [137, 62, 48, 50, 51, 29, 132] by allowing non-unit edge weights as well as a restriction on the number of edges that may be colored with a single color. The only algorithm that is as general as **A1** - **A3** is **KT** [9, 10], and as shown in this chapter, our algorithms out-perform it both in terms of theoretical and experimentally-measured performance.

3.9.2 Conclusions and future work

A key question that arises at this point is: which algorithm should be used for data transfer scheduling in bipartite architecture graphs, and under which operating conditions? Our theoretical and experimental results show that for all the situations considered in this chapter, either **A1** or **A2** should be used over the previous best algorithm, **KT**. In general, **A2** is the algorithm of

choice unless either the maximum transfer length K is greater than a small constant, or the number of communicating entities (disks, transmitters, etc) n is very high, in which cases **A1** should be used.

It is interesting to consider the poor observed performance of **A3**. This seems to be because of two reasons. Firstly, the conditions under which its experimentally measured performance was better than **A2** were very limited: a relatively small number of transfers m, of large lengths K, to be carried out between relatively few entities n, with a high degree of parallelism k. This is because the constants in **A3**'s asymptotic time complexity seem to be very high. Secondly, the theoretical worst-case space complexity of **weighted-edge-color**, and hence **A3** is very large: $O(mn \log K)$ [51]. Our experiments show that in fact even on average the amount of space required is unacceptably high for most situations of practical interest. For instance, 32 MB of main memory were not sufficient to handle input graphs with parameters greater than $n = 64, k = 4, m = 1000$, and $K = 80$. Performance considerations aside, it was found that **A3** was more difficult and time-consuming to implement than any of the other algorithms, and consisted of about twice as many lines of C code. We conclude that although the algorithmic technique underlying **weighted-edge-color** is elegant and of theoretical interest, it does not lead to a practical algorithm for our application.

For future work, there are two interesting questions. The first is whether Gonzalez and Sahni's algorithm [62] can indeed be extended to solve DTS, either by using the k-filling technique or by some other means, and if so, whether its performance when implemented is fast enough to make it an attractive practical solution to DTS for interesting applications. The second

is whether k-filling is needed at all for optimal k-coloring of bipartite graphs, i.e., perhaps this constraint can be satisfied at a lower level in the coloring algorithm, say at the level of finding augmenting paths.

To summarize our contributions in this chapter, we have developed and experimentally evaluated three new algorithms for scheduling data transfers in communications and parallel computer systems. These algorithms apply to a significant class of applications, such as satellite data transfers and parallel I/O due to 3D visualization software. The algorithms apply to a common class of architectures, including satellite TDMA switches, and shared-bus multiprocessors such as the Sequent [100], Encore Multimax [143] and the IBM RP3 [115]. Our theoretical and experimental investigations show that our algorithms perform significantly better than the previous best available algorithm, **KT**. Our algorithms also generalize previous theoretical work on edge-coloring algorithms for bipartite graphs [137, 62, 48, 50, 51, 29, 132], both by allowing weighted edges and restrictions on the number of edges that may be colored with a single color. Finally, to our knowledge, ours is the only extensive experimental study of the behavior of four edge-coloring algorithms for bipartite graphs in which the effects of varying the problem parameters are investigated systematically. Such experimental studies are very valuable from a practical standpoint; for example, we have shown that the practical usefulness of the **weighted-edge-color** algorithm [51] used in **A3** is severely limited both in terms of space and time cost.

Chapter 4

Heuristics for Scheduling in Bus Architectures and TDM Switches

In the previous chapter we discussed optimal algorithms for scheduling data transfers in shared-bus multiprocessors and single TDMA switches. While the algorithms we developed, **A1 - A3**, are in general faster than previous optimal algorithms like **KT** [10], we would like to have even faster algorithms, since in most applications scheduling algorithms are executed repeatedly, and any gain in speed helps overall system performance.

In this chapter we turn our attention to approximation algorithms (or heuristics) for the *SimpleDTS* and *DTS* problems restricted to unit-length transfers. Two simple greedy heuristics for unit-length transfers, **HDF** and **HCDF**, have been proposed and experimentally evaluated by Somalwar [132]. In graph-theoretic terms, these heuristics are essentially approximation algorithms for edge-coloring bipartite graphs with unit-weight edges. Both heursitics performed well in experiments, both with random input graphs as well as input graphs simulating the projected parallel I/O workload generated by applications such as 3D visualization and split-step migration. For instance,

for experiments using random bipartite graphs as inputs, one of the heuristics always generated the exact solution, while running in less than 10% of the time taken by an optimal algorithm [132].

While Somalwar's result is encouraging, it is obvious that experimental evaluation can only examine a small number of combinations of the input parameters, and thus explore only a tiny fraction of the input space. In this chapter we present the first analysis of the worst-case execution time of the heuristics, as well as an analysis of their divergence from the optimal solution in the worst case. We will quantify the divergence from the optimal solution by finding a performance guarantee for each algorithm, defined as follows.

Def. If an approximation algorithm produces a schedule of length $L'(RG)$ for a problem instance with resource graph RG, and $L(RG)$ is the optimum schedule length, then the *performance guarantee* of that algorithm is $P(n)$, where $P(n)$ is the maximum value of the ratio $L'(RG)/L(RG)$, over all RG with at most n vertices.

(For the algorithms in this chapter, we will be actually be interested in $P(d)$, where d is the degree of RG.) In sec 4.1 we derive a bound for the worst-case time complexity and the performance guarantee for Somalwar's heuristics [132]. We remark that this bound is tight, i.e., it is possible to systematically generate graphs for which the heuristic performs as badly as the worst-case. In section 4.3 we compare Somalwar's experimental results [132] with the theoretical results, and finally we end with a discussion.

4.1 The Highest Degree First (HDF) Heuristic

We define the $Unit - SimpleDTS$ problem to be the $SimpleDTS$ problem restricted to the case where all tasks have unit length, i.e., for all $t \in T$, $Lp(t) = 1$. By the observations made in the previous chapter, in graph-theoretic terms, $Unit - SimpleDTS$ corresponds to the problem of finding a minimum edge-coloring of a bipartite graph with unit-weight edges. Similarly, $Unit - DTS$ corresponds to finding a minimum edge-coloring for a bipartite graph with unit-weight edges given that at most $k \leq n$ edges may be colored with a single color. We analyze the behavior of **HDF** first for $Unit - SimpleDTS$, and then for $Unit - DTS$.

4.1.1 The $Unit - SimpleDTS$ Problem

We introduce some additional terminology. Vertices $a, b \in A \cup B$ of a graph $G = (A, B, E)$ are called *partners* if $(a, b) \in E$. A vertex is said to be *colored with color c* if some edge incident upon it is colored c. Note that an edge has a unique color but a vertex may have multiple colors. A vertex is said to be *fully colored* if every edge incident upon it is colored. The degree of a vertex v is denoted $d(v)$. The degree of the graph G by $d(G)$, or simply d if clear from context. A sequence is denoted by angle brackets.

Somalwar's Highest-Degree-First **HDF** heuristic for $Unit - SimpleDTS$ for a graph $G = (A, B, E)$ is specified as algorithm **HDF** below. The *Sort-by-degree()* procedure sorts the vertices in order of descending degree. The

"break" statement exits the smallest enclosing loop. The basic idea of **HDF** is give priority to coloring the vertices in order of their degree.

Algorithm HDF

Input: Bipartite graph $G = (A, B, E)$

Output: An edge-coloring of G

```
1. ⟨v_1, ..., v_n⟩ := Sort-by-degree(A ∪ B);
2. while E ≠ {}
3.     E' := {};
4.     for each v read in sequence from ⟨v_1, ..., v_n⟩ {
5.         for each e = (v, w) ∈ E {
6.             if e is not adjacent to any edge in E' {
7.                 Add e to E' and remove it from E
8.                 Reduce degree of v, w by 1 and
                       remove from ⟨v_1, ..., v_n⟩
9.                 break
10.            }
11.        }
12.    }
13.    Color all edges in E' with a new color
14.    ⟨v_1, ..., v_n⟩ := Sort-by-degree(A ∪ B);
15. }
16. end
```

Recall that the minimum number of colors, or schedule length, is d for a bipartite graph of degree d. How many colors will **HDF** use in the worst

case? It is useful to answer a more general question instead: how many colors will a greedy heuristic use in the worst case? We first specify the *greedy heuristic* which, for every color, attempts to color as many edges as possible with that color.

Algorithm Greedy Heuristic
Input: Bipartite graph $G = (A, B, E)$
Output: An edge-coloring of G

```
1. Assign some order F = ⟨e_1, e_2, ..., e_m⟩ to the edges of E
2. i := 0
3. while F ≠ {} {
4.     for each e read in sequence from F {
5.         if e can be colored with color i {
6.             color e with color i
7.             remove e from E and F
8.         }
9.     }
10.    i := i + 1
11. }
```

Clearly, any execution of **HDF** can be repeated by the greedy heuristic by choosing an appropriate initial ordering of the edges. Thus **HDF** is a special case of the greedy heuristic. We now state a simple but useful fact.

Lemma 4.1 *At the end of iteration i of the* **while** *loop of the greedy heuristic, if some vertex v is not fully colored and is not colored i, then all of v's partners are colored i.*

proof. Suppose not. Then v as well as at least one of its partners, say w, is not colored i. But then the greedy heuristic would have colored the edge (v, w) with i. □

Lemma 4.2 *The greedy heuristic produces a coloring using at most* $2d - 1$ *colors for a bipartite graph of degree* d.

proof. For a vertex v let $deg(v)$ be its degree and $P(v)$ its set of partners. Let $L(v) = deg(v) + \max\{deg(w) : w \in P(v)\}$. From Lemma 4.1, every color used by the greedy heuristic reduces $L(v)$ by at least 1. A special case occurs for the last color used to color v; for this case, there is one remaining edge incident on v, so that when it is colored $L(v)$ is reduced by 2. Therefore at most $L(v)-1$ colors are used to color all edges incident to v. Since $L(v) \leq 2d$, the result follows. □

It can be shown that this bound is tight for **HDF**. A simple example where the $2d - 1$ bound on schedule length is met for $d = 2$ is the four-transfer example given in Chapter 1. However, we can prove that such an example bipartite graph $G(d)$ can be constructed for any positive integer d. In fact, we will show that $G(d)$ is a tree. We first introduce some notation and definitions; see Fig. 4.1 for examples of their use.

Notation. Upper-case italic letters denote vertices or subtrees of a tree; they may be subscripted. If A and B are vertices, $A;B$ denotes that they are siblings, and $A|B$ denotes that A is the parent of B. The letter R may

be used to distinguish the root of a (sub)tree, and C for its child. Thus $R\langle C_1; C_2\rangle$, where the C_i are distinguished vertices, denotes a binary tree of two levels. A set of identical siblings is denoted using an array notation: thus $R\langle C[2]\rangle$ also denotes a binary tree with two levels. Angle brackets have higher precedence than semi-colons. Thus $R\langle C_1; C_2\rangle \ ; A$ denotes a forest with two trees, and $R\langle C_1; C_2; A\rangle$ denotes a ternary tree with two levels.

Def. Two trees S and T with roots R_S and R_T, respectively, are *root-merged* by deleting R_S and R_T (along with any incident edges), introducing a vertex R, and adding edges from R to every child of R_S and R_T. Using the notation above, and letting + denote root-merging, let

$$S = R_S\langle S_1; S_2; ...; S_i\rangle$$

$$T = R_T\langle T_1; T_2; ...; T_j\rangle$$

Then $S + T = R\langle S_1; ...; S_i; T_1; ...; T_j\rangle$.

We now construct two families of trees to be used later in the construction of $G(d)$, and consider how they could be colored.

Def. The tree $F(i, d)$, with $d > 1$, is defined mutually recursively with the tree $H(i, d)$ as follows. See Fig. 4.1 for examples.

1. $F_{0,d}$ consists of a single vertex.

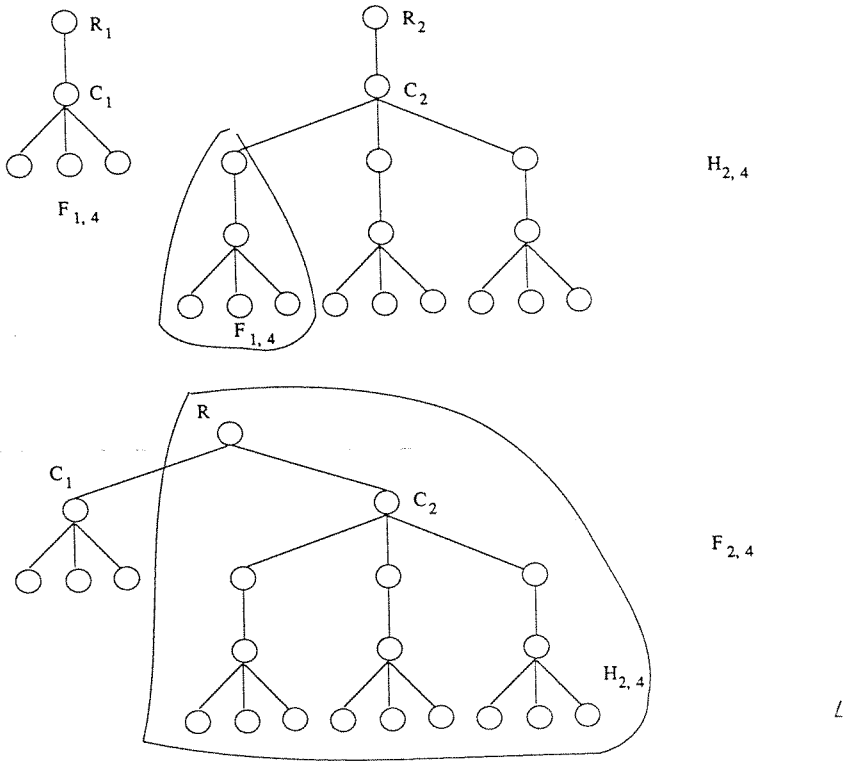

Figure 4.1: Example construction to show **HDF** takes up to $2d-1$ colors to color a graph of degree d

2. $H_{1,d} = R_1\langle C_1\langle F_{0,d}[d-1]\rangle$
3. $F_{1,d} = H(1,d)$
4. For $1 < i < d$, $H(i,d) = R_i\langle C_i\ \langle F_{i-1,d}[d-1]\rangle\rangle$
5. For $1 < i < d$, $F_{i,d} = H_{1,d} + H_{2,d} + ... + H_{i,d}$

$\langle\rangle$ has precedence over $+$, which has precedence over ;. Observe that for every tree $H_{i,d}$, the child of the root, C_i, has degree d, i.e., is critical. Also note that for every $F_{i,d}$, the children of its root are critical.

Lemma 4.3 *For every tree $F_{i,d}$, $0 < i < d$, there exists a sequence of choices*

made by **HDF** *such that the root of* $F_{i,d}$ *is colored with every color in the set* $\{1, ..., i\}$.

proof. By induction over j.

base. $j = 1$. For $F_{1,d} = H_{1,d} = R_1\langle C_1\langle S[d-1]\rangle\rangle$, the choice of coloring edge (R_1, C_1) with color 1 suffices.

hyp. For all $F_{i,d}$, $0 < i < j < d$, there exists a sequence of choices made by **HDF** such that the root of $F_{i,d}$ is colored with all colors in {1, ..., i}.

ind. Consider the coloring of $F_{j,d}$. By definition,

$$F_{j,d} = H_{1,d} + H_{2,d} + ... + H_{j,d}$$

$$= R_j\langle C_1\langle F_{0,d}[d-1]\rangle;\ C_2\langle F_{1,d}[d-1]\rangle;\ ...\ C_j\langle F_{j-1,d}[d-1]\rangle;\ \rangle$$

We will show that there is a sequence of choices made by **HDF** such that for all $i, 0 < i \leq j$, edge (R_j, C_i) in the expression above is colored by color i. First note that by the hypothesis, for every i, $0 < i < j$, there exists a sequence of choices s_i such that the root of $F_{i,d}$ is colored with all colors in $\{1, ..., i\}$. Clearly, it is possible to merge these sequences appropriately so that the resulting sequence, s, colors the root of every $F_{i,d}$ with all colors in $\{1, ..., i\}$. We now show how s is extended by a sequence of choices that can be made by **HDF**.

Since every C_i is critical, and uncolored, it is eligible to be chosen by **HDF**. Let the first choice be to color C_1. By applying the hypothesis the root of every $F_{0,d}$, is not colored; let **HDF** choose to color C_1 by coloring (R_j, C_1) with the color 1. Now let **HDF** choose to color C_2. By the hypothesis, the root of every $F_{1,d}$ is colored with color 1; also, R_j was just colored 1. Therefore C_2 cannot be colored using color 1. Let **HDF** choose to color C_2 by coloring edge (R_j, C_2) with color 2.

HDF continues to choose to color each C_i in turn by choosing to color (R_j, C_i) with color i. The sequence of **HDF**'s choices is concatenated to the sequence s, and this gives the result. □

Theorem 4.1 *For any positive integer d, there exists a bipartite graph of degree d and a sequence of choices made by* **HDF** *such that* **HDF** *uses* $2d-1$ *colors to color the graph.*

proof. By construction of the graph. Let

$$G(d) = R\langle F_{d-1,d}[d]\rangle$$

From Lemma 4.3 there exists a sequence of choices made by **HDF** such that the root of every $F_{d-1,d}$ is colored with all colors in $\{1, ..., d-1\}$. Therefore, each of the links incident to R will have to be colored with a color not in the set $\{1, ..., d-1\}$, and each will require a distinct color. Therefore d colors are required to color the links incident to R in addition to the colors $\{1, ..., d-1\}$,

i.e., at least $d + d - 1 = 2d - 1$ colors are required to color $G(d)$. By Lemma 4.2 we know that **HDF**, being a greedy algorithm, requires at most $2d - 1$ colors to color $G(d)$. It follows that there exists a sequence of choices for which **HDF** uses $2d - 1$ colors to color $G(d)$. □

To summarize, by Lemma 4.2 we have shown that any greedy heuristic, and hence **HDF**, uses at most $2d - 1$ colors to color a bipartite graph, and by Theorem 4.1 we have shown that for any d we can construct an example for which **HDF** actually uses $2d-1$ colors. We can use Lemma 4.2 to also obtain the time complexity of **HDF**.

Theorem 4.2 **HDF** *takes time* $O((m + n)d)$ *to solve* $Unit - SimpleDTS$, *and produces a schedule of length at most* $2 - \frac{1}{d}$ *times the optimal length.*

proof. For the time complexity, note that *Sort-by-degree()* takes time $O(n+d)$, if a bucket sort is used. For each color, each edge is examined at most once. Thus each color (i.e., iteration of the while loop) takes time $O(m + n + d)$ [132]. Using Lemma 4.2, the total time is thus $O((m + n + d)(2d - 1)) = O((m + n)d)$. The performance guarantee follows from Lemma 4.2 also. □

4.1.2 The $Unit - DTS$ Problem

The program for implementing the **HDF** heuristic for handling the situation where a color may be used to color at most $k \leq n$ edges is a slight modification of the program given above. We add the following line,

11.5 if $|E'| = k$, break

That is, after every edge is added to E', the program checks if $|E'| = k$, and if so, does not add any more edges to E' for the current color. We call this program **MHDF** for convenience.

The analysis of **MHDF** is slightly more complicated than for **HDF**. In particular, we are not able to prove a bound that is tight for all inputs.

Lemma 4.4 **MHDF** *produces a coloring using at most* $\lfloor m/k \rfloor + (2d-1)$ *colors for a graph of* n *vertices,* m *edges, and degree* d*, if at most* $k \leq n$ *edges may be colored with a single color.*

proof. Suppose **HDF** was used on the graph instead of **MHDF**. In the worst case it uses $2d-1$ colors, with color i coloring m_i edges, and $\sum_{i=1}^{2d-1} m_i = m$. If **MHDF** uses the same sequence of choices, coloring m_i edges may require up to $\lfloor m_i/k \rfloor + 1$ colors. The result follows. □

Note that this bound is exact for the graph $m = n = 5$, $d = 1$, $k = 2$. On the other hand, it is not tight for the class of graphs with $k = n$ and $k = 1$. In the former case, **MHDF** is simply **HDF**, so at most $2d-1$ colors are needed. In the latter case, exactly $m/k = m$ colors are needed.

4.2 The Highest Combined Degree (HCDF) Heuristic

An obvious modification to the **HDF** heuristic is to give preference not necessarily to vertices of highest degree, but to edges whose end vertices have the highest combined degree. Somalwar [132] experimentally investigated this Highest-Combined-Degree-First heuristic, **HCDF**, and found that it gave optimal or near-optimal solutions for input graphs generated randomly. In this section we find the time complexity and performance guarantee of **HCDF**, for both the $Unit - SimpleDTS$ and $SimpleDTS$ problems. This turns out to be a simple extension to the analysis for **HDF**.

Our first observation is that **HCDF** is also a special case of the greedy heuristic. We thus obtain the following result.

Theorem 4.3 **HCDF** *takes time* $O((m+d)d)$ *to solve* $Unit - SimpleDTS$, *and produces a schedule of length at most* $2 - \frac{1}{d}$ *times the optimal length.*

proof. The performance guarantee follows from Lemma 4.2 since **HCDF** is also a greedy heuristic. For the time complexity, we observe that the edges can be sorted using a bucket sort, with buckets in the range 1 to $2d$, and for every color, each edge is examined at most once. Therefore every color takes time $O(m+d)$, and there are at most $2d-1$ colors used. □

We can also show that for certain graphs exactly $2d-1$ colors will be used by modifying the construction used for **HDF**.

Theorem 4.4 *For any positive integer d, there exists a bipartite graph of degree d and a sequence of choices made by* **HCDF** *such that it uses $2d-1$ colors to color the graph.*

proof. Construct $G(d)$ as described for Theorem 4.2. We will construct a new tree $T(d)$ by modifying $G(d)$; initially set $T(d) = G(d)$. We call an edge *critical* in $G(d)$ or $T(d)$ if the combined degree of its end vertices is $2d$. For all $i \in \{1, ..., d\}$, edge $rc(i, d)$ in $G(d)$ has at least one critical vertex; for each such edge, ensure the corresponding edge in $T(d)$ has both end vertices critical by adding appropriate edges and vertices if necessary. Now the sequence of edge-colorings used by **HDF** to color $rc(i, d)$, $i \in \{1, ..., d\}$, can be used by **HCDF**. Thus it takes $d-1$ colors before the root of $T(d)$ is colored, and an additional d colors to color all the edges incident to the root. Since **HCDF** is greedy, by Lemma 4.2 all other edges can be colored using $2d-1$ colors. □

Finally, we call the algorithm **HCDF** modified to color at most $k \leq n$ edges with the same color the Modified-HCDF algorithm, **MHCDF**. For this algorithm applied to the $Unit-DTS$ problem, we can obtain a result similar to **HDF**.

Lemma 4.5 **MHCDF** *produces a coloring using at most $\lfloor m/k \rfloor + (2d-1)$ colors for a graph of n vertices, m edges, and degree d, if at most $k \leq n$ edges may be colored with a single color.*

proof. Similar to Lemma 4.4. □

4.3 Comparison of experimental and theoretical results

Somalwar [132] has experimentally evaluated the performance of **HDF** and **HCDF** on instances of the $Unit-DTS$ problem He compared their behavior to that of the exact algorithm **A**, which is a special case of **A2** for unit-weight edges implemented by Somalwar [132]. We also compare their behavior to the performance guarantees derived in this chapter.

Somalwar performed this experimentation in the context of the parallel I/O application, i.e., scheduling parallel I/O operations for a shared-bus multiprocessor system. For this context, certain parameters of the input graph $G = (A, B, E)$ were fixed. In particular, it was chosen that $n_1 = |A| = 16$, $n_2 = |B| = 64$, $m = |E|$ varied from 100 to 1000, and k varied from 4 to 16. Edges were generated using a pseudo-random generator.

Somalwar [132] found that in his experiments both **HDF** and **HCDF** produced schedules that are almost always of the optimal length. In that sense, they perform much better on average, for this set of experiments, than their performance guarantees predict for their worst case behavior. Although these results seem surprisingly good, there are some recent related experimental results to support them. Moret [105] found that for maximum cardinality bipartite matching, using a simple greedy heuristic similar to **HDF** delivered an optimal solution at least 99% of the time. Since the basic operation of edge-coloring unit-weight graphs can be regarded as repeated matching, these results are consistent with Somalwar's observations.

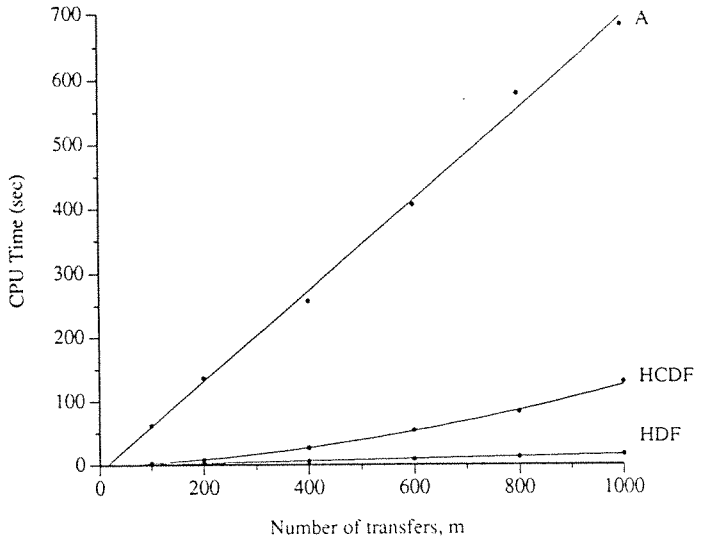

Figure 4.2: CPU time versus number of transfers for $k = 4$

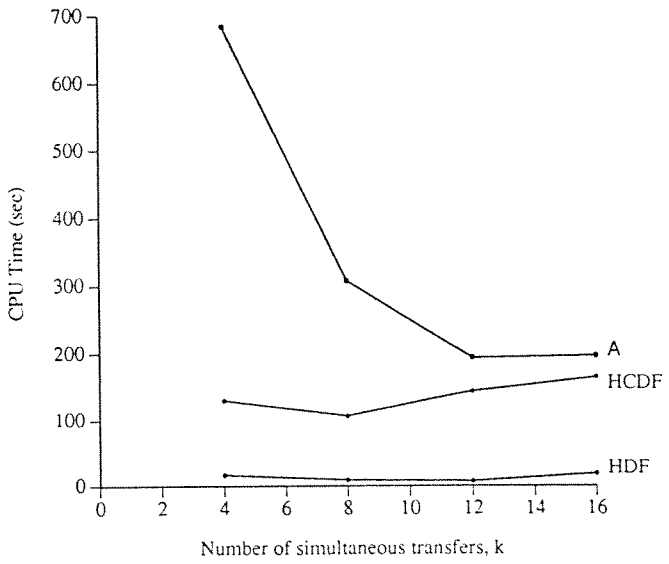

Figure 4.3: CPU time versus number of simultaneous transfers possible for $m = 1000$

In Fig. 4.2 and Fig. 4.3 the execution time of **A**, **HDF** and **HCDF** is compared, as m and k are varied respectively. It can be seen that **HDF** executes in at most 10% of the time required for **A2** for this set of experiments. On the other hand, **HCDF** may take up as much as 80% of the time required by **A2**. Since both heuristics almost always produce optimal schedules, this set of experiments indicates that **HDF** is to be preferred over **HCDF** and **A2**, unless it is essential to produce optimal schedules, in which case **A2** should be used.

4.4 Discussion

4.4.1 Previous related work

The literature on exact edge-coloring of bipartite graphs with unit-weight edges was discussed in the previous chapter. To our knowledge, there has been no previous work on analysis of approximate edge-coloring of bipartite graphs with unit-weight edges.

The only related work that may be relevant deals with approximate edge-coloring of general graphs and multigraphs. Holyer [70] showed that determining whether a general graph can be colored in d or $d+1$ colors (the *classification problem* [41]) is NP-complete. A consequence of his result was that, unless P = NP, there does not exist an approximation algorithm for edge coloring a general multigraph using at most $(4/3-\epsilon)D$ colors, for any $\epsilon > 0$, where $D \in \{d, d+1\}$ is the minimum number of colors required. This

would seem to imply that for general multigraphs finding a provably good approximation algorithm is difficult. However, an algorithm using no more than $(4/3)D$ colors, and running in time $O(m(n + D))$ was developed [69]. This algorithm uses an "interchange approach" as its basis: for each edge, check if some "simple" recoloring of the colored edges would eliminate the need for an additional color. As the authors say, "in order to prove better bounds, the 'simple' recolorings become more complicated" [69].

We note three deficiencies with this interchange heuristic. Firstly, it does not consider the practical constraint of a limited number of simultaneous data transfers, i.e., $k \leq n$. For our applications, this constraint corresponds to the realistic situation of limited bus bandwidth or switching capacity being available in the system. Secondly, although this heuristic has a better performance guarantee than those analyzed in this chapter, its time complexity is slightly worse, unless $d = n$. Thirdly, since the **HDF** and **HCDF** heuristics do not perform any backtracking or recoloring, they are likely to have smaller constants for their time complexity, and are likely to be simpler to implement, than the interchange heuristic.

To our knowledge, there has been no previous published experimental evaluation of approximation algorithms for edge-coloring bipartite graphs other than Somalwar [132]. The only unpublished results we are aware of are those by Moret [105], which provide evidence to support some of Somalwar's results.

4.4.2 Conclusions and further work

Approximate algorithms for edge coloring bipartite graphs are very attractive from a practical standpoint, particularly if they are to be used as the basis of scheduling algorithms that are executed very frequently in a data transfer application. While Somalwar [132] has suggested some simple greedy heuristics that seem to perform extremely well when evaluated experimentally, there was no analysis of the time complexity or the performance guarantee for these heuristics.

Our contribution in this chapter has been to prove the time complexity and performance guarantees for the heuristics proposed by Somalwar. The heuristics apply to the data transfer problem DTS when restricted to unit-length transfers. We have shown that the heuristics generate schedules less than twice the length of the optimal schedule, in the worst case, and take at most time $O((m+n)d)$ to execute, where n is the number of vertices, m the number of edges and d the degree of the input graph.

Our result enables us to compare the performance of Somalwar's heuristics with more sophisticated "interchange" heuristics [69]. The interchange heuristics are more general than those of Somalwar as they are applicable to general graphs as well as multigraphs. They also provide a better performance guarantee for only a slight increase in time complexity. However, we observe that for our applications, the Somalwar heuristics may be preferable as the applications are restricted to bipartite graphs, and the heuristics allow the case when a limit is placed on the number of simultaneous data transfers allowed.

We also surmise that the interchange heuristics have larger time constants, worse average-case execution time, and are harder to implement, than the simple greedy heuristics of Somalwar.

For future work, we suggest two questions. The first question is to experimentally and theoretically compare the interchange heuristics and Somalwar's greedy heuristic, and determine the range of parameters for which each is suitable. The comparison could include theoretical average-case analysis (e.g. [46]) as well as careful experimental evaluation. The second question is to consider the possibility of using parallel algorithms for scheduling; these may be especially suitable for the parallel I/O application. The results of Karloff and Shmoys [86] provide a starting point in this direction.

Chapter 5

Scheduling in Hierarchical Architectures

We have so far discussed scheduling of data transfers in system architectures which have a rather simple, although common, structure. The architecture of the *DTS* and the *SimpleDTS* problems assume that there is a direct dedicated link between every sender and every receiver; constraints arise in the number of links that may be used simultaneously, and in the capacity of each sender and receiver to engage in at most one transfer at any given time. In the context of the parallel I/O application, this architecture corresponds to the commercially succesful and popular class of shared-bus multiprocessors, in which processors and disks are connected by a set of parallel buses. For the communications application, it corresponds to the case of scheduling transfers through a single TDM switch, which can be viewed as a single multiplexer feeding a single demultiplexer.

In this chapter we consider more complex architectures that do not assume a direct dedicated link between every sender and every receiver. In particular, we consider a system where the data transfers must pass through a hierarchically arranged set of communication paths, which form a communication

tree (see Fig. 5.1). We also generalize the architecture to allow arbitrary capacities, drawn from the set of positive integers, for each link in the architecture. Note that this also does away with the restriction that a sender or receiver can engage in at most one transfer in any given time. We call this architecture the tree architecture.

In section 5.1 we define the tree architecture formally in our model and specify the data transfer scheduling problem that we are interested in. In section 5.2 we show that this problem has application in three different areas: parallel I/O, switching systems, and file transfer in computer networks. The tree architecture is also of theoretical interest: we surmise that if the architecture is made more complex than a tree, optimal preemptive scheduling of integer-length transfers without precedence constraints cannot be done in polynomial time.

We then develop, in section 5.3, the outline of an algorithm to optimally solve the scheduling problem, and in section 5.4 we show how the algorithm can be designed to obtain a solution in time $O(Cn^4)$, where n is the number of senders and receivers, and C is the average number of transfers a sender or receiver can engage in at any one time. This algorithm, which we call the **Tree** scheduling algorithm, has been presented in [77, 125]. In general, **Tree** is a generalization and improvement in time complexity over previous algorithms [24, 11, 96, 20, 136] for this class of problems.

We have implemented the **Tree** algorithm. In section 5.5 we report the results of an experimental evaluation of the behavior of the algorithm for random

input instances. Finally, in section 5.6 we discuss previous related work, and end with some conclusions.

5.1 Definition of the problem

We define the generalized data transfer scheduling problem *TreeDTS*, which differs from *DTS* in allowing tree-structured interconnection networks of the type shown in Fig. 5.1. The interconnection network is defined formally as *AG* below.

$TreeDTS = (PG, AG, RG, f, Preempt)$

where $Preempt = true$, f is makespan, and, PG, AG, and RG are defined as follows.

$PG = (T, Ep, Lp)$ with $|T| = m$, $Ep = \{\}$, and $Lp(t) \geq 0$ for all $t \in T$ are the task lengths.

$AG = (R, Ea, La)$ and (see Fig. 5.1) $RT = \{SUSER, RUSER, MUX, DMUX, NULL\}$; $|R| = n + n' + 1$, n is the number of resources of type $SUSER$ or $RUSER$, n' is the number of type MUX and $DMUX$, and there is one resource of type $NULL$. Ea is a directed tree whose root is the resource of type $NULL$. The root has a left subtree MT called the multiplexer subtree with interior nodes of type MUX, leaves of type $SUSER$, and arcs directed towards the root. (The definition of the right subtree DT follows by analogy).

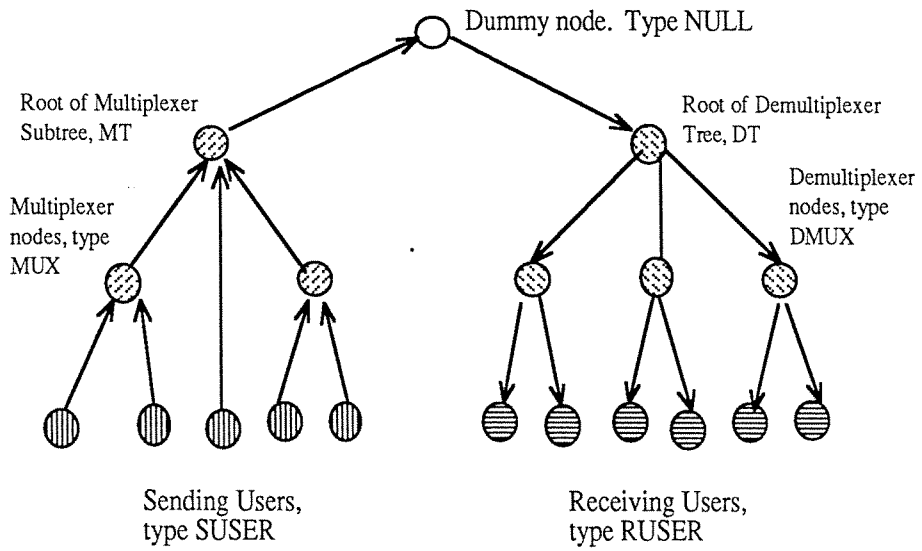

Figure 5.1: Model of a tree-structured architecture

Now $La(r) = 0$ if $r \in R$, and $La(e)$ is the capacity of arcs e (in packets per second) for $e \in Ea$. We assume all interior vertices have degree at least 3 so as to avoid degenerate trees. We also assume that for vertices in MT the sum of $La(e)$ for incoming arcs e is at least the value of $La(e')$ for the single outgoing arc e'. There is an analogous assumption for DT, while for the root the capacity of the incoming arc equals that of the outgoing arc.

$RG = (R, Er, Lr)$ is a bipartite graph, where Er is a set of arcs from vertices of type $SUSER$ (senders) to those of type $RUSER$ (receivers) representing the data transfer operations to be scheduled, and Lr is a bijection from Er to T. Note that since there is a unique path betwen each sender-receiver pair, only the assignment of tasks to senders and receivers is shown in RG, the assignment of other resource types being left implicit.

5.2 Three Practical Applications

By casting the extended data transfer scheduling problem in our model, we see that it is a generalization of problems studied for three applications: parallel I/O, satellite switching, and network file transfers [80, 81]. Thus the results we derive in this chapter are available to all three applications. We describe these applications below.

Application 1: Parallel I/O in multiprocessor systems

In this chapter we consider cases of the I/O scheduling problem in which we do not assume a direct dedicated link from every processor to every memory. In addition, a processor or memory is not limited to engaging in at most one transfer at any given time. This problem is applicable to I/O scheduling in a variety of tree-structured parallel computer architectures, as well as interconnection networks such as KYKLOS [102].

Parallel database machines have been built that have tree-structured architectures, such as the VLSI tree machine of Song [133] and the relational database machine REPT [127]. The VLSI tree machine consists of two mirrored binary trees connecting a common set of leaves. The root of the top (called "circle") tree receives data and commands from the external host and broadcasts them down to the leaves ("square nodes"). The leaves perform the data manipulations in parallel and deliver results via the bottom ("triangle") tree. The interior nodes of the bottom tree combine results from the leaves before transferring them to the external host via the root of the bottom tree.

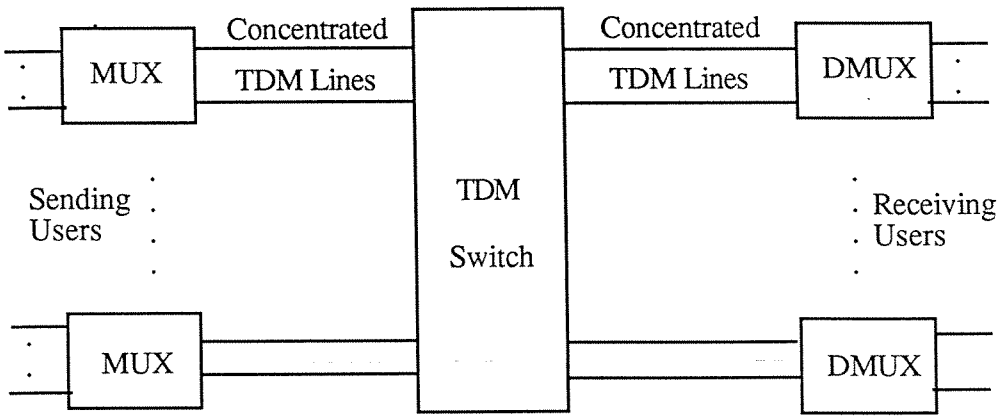

Figure 5.2: SS/TDMA hierarchical switching system

Application 2: Hierarchical switching systems

Hierarchical time division multiplexed (TDM) switching systems have been proposed [39] that connect sending users via a bank of multiplexers, followed by a TDM switch, followed by a bank of demultiplexers to receiving users, as in Fig. 5.2. Hence, the switch has three stages. Advantages of the hierarchical structure are that fewer switches may be needed to serve the user population; trunking efficiency may be increased due to the fact that end users have access to multiple input links; and modular growth is possible if additional lines and multiplexers/demultiplexers are required.

The hierarchical switching systems we consider in the remainder of this paper have an arbitrary number of stages, but are required to conform to a tree topology. A special case of our switching systems includes the one in Fig. 5.2

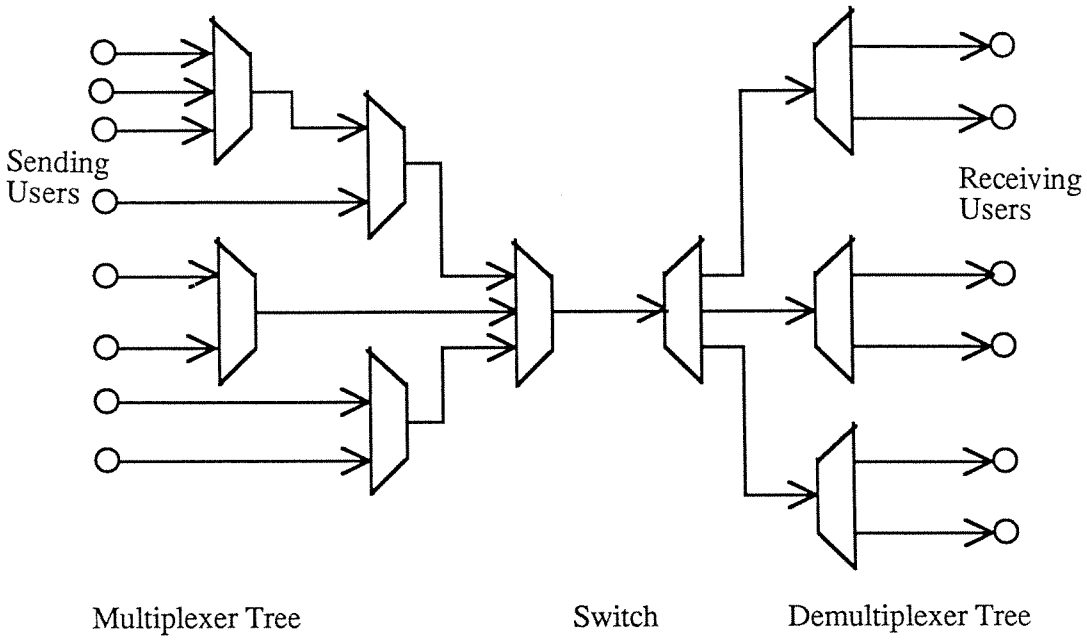

Figure 5.3: Hierarchical switching system

but where the middle $n \times n$ switch can transport at most $k \leq n$ packets per slot.

Note that the central TDM switch can also be regarded as a multiplexer feediing a demultiplexer. Thus the switching system can be regarded as being composed of only two elements, as shown in Fig. 5.3. We will use this model for the system architecture in the rest of this chapter.

Application 3: File transfers in computer networks

Consider a communications network where each node has several ports that can simultaneously transfer files, file transmission can be preempted, and the maximum number of simultaneous transfers at any time in the network is fixed. Each user is connected to a single network node and assigned a unique

port on that node for file transfers. Such a network can be modeled as a hierarchical switching system, or a specialized tree-structured architecture (a special case of the architecture shown in Fig. 5.1).

5.3 The Tree scheduling algorithm

In order to develop an algorithm for solving *TreeDTS* optimally, we first introduce some notation and some definitions. We then introduce the notion of critical transfers and develop the outline of an algorithm.

5.3.1 Basic definitions and notation

The sets of senders and receivers are denoted V_S and V_R, and the sets of multiplexers and demultiplexers are denoted V_M and V_D. The architecture graph is sometimes called a system graph in this section, and is shown in Fig. 5.4. It is denoted (V, E) where $V = V_S \cup V_M \cup V_D \cup V_R$ and the directed links E are as follows. The set $E = E_M \cup E_D \cup \{[x_M, x_D]\}$, where $x_M \in V_M$, $x_D \in V_D$, $(V_S \cup V_M, E_M)$ denotes the multiplexer tree with root node x_M, and $(V_R \cup V_D, E_D)$ denotes the demultiplexer tree with root node x_D. In addition, the links of $(V_S \cup V_M, E_M)$ are directed towards x_M, and the links of $(V_R \cup V_D, E_D)$ are directed away from x_D. Note that a multiplexer node $v \in V_M$ (resp., demultiplexer node $v \in V_D$) is one with a single outgoing (resp., incoming) link and at least two incoming (resp., outgoing) links. Let $n = |V_S| + |V_R|$.

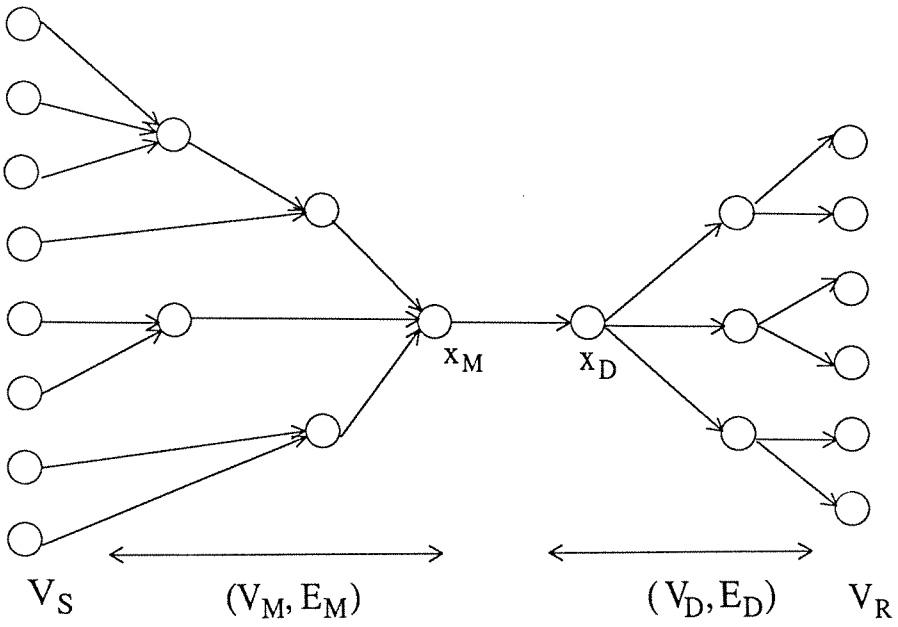

Figure 5.4: Switching system graph (V, E)

For each link $e \in E$, the positive integer $c(e)$ is the number of transfers that can be carried on simultaneously over that link; in terms of the satellite switching application, it is the number of packets that can be delivered over the link in one time slot. For each node $v \in V$, let $I(v)$ and $O(v)$ be the sets of incoming and outgoing links of v, respectively. We assume for a multiplexer node $v \in V_M$, $\sum_{e \in I(v)} c(e) \geq \sum_{e \in O(v)} c(e)$; and for a demultiplexer node $v \in V_D$, $\sum_{e \in I(v)} c(e) \leq \sum_{e \in O(v)} c(e)$. For a user node $v \in V_S \cup V_R$, let $C(v)$ be the capacity $c(e)$ of the single link e incident to v.

Note that $|V| \leq 2n$, since $(V_S \cup V_M, E_M)$ and $(V_R \cup V_D, E_D)$ are trees and each node in V_M (resp., V_D) has at least two incoming (resp., outgoing links) links.

We now introduce some specializations of the definitions given in Chapter 2,

in order to simplify the discussion of the solution to *TreeDTS*. Let $E_{SR} = \{[u,v] : u \in V_S, v \in V_R\}$. We call any nonnegative integer matrix $r = (r(e) : e \in E_{SR})$ a *traffic matrix*. The interpretation of $r([u,v])$ is the number of packets to be transferred through the system from user u to user v; in effect it captures the resource graph of the problem. If r is a traffic matrix then let $t_r = (t_r(e) : e \in E)$, where $t_r(e)$ is the number of packets to be transferred over link e with respect to r. More formally, t_r satisfies the following: if e is the outgoing (resp., incoming) link of node $v \in V_S$ (resp., $v \in V_R$) then $t_r(e) = \sum_{u \in V_R} r([v,u])$ (resp., $\sum_{u \in V_S} r([u,v])$); and for all nodes $v \in V_M \cup V_D$, $\sum_{e \in I(v)} t_r(e) = \sum_{e \in O(v)} t_r(e)$. Clearly, t_r can be calculated from r in $O(n^2)$ time.

A traffic matrix r is called a *feasible transfer* if for all $e \in E$, $t_r(e) \leq c(e)$. A *schedule* is a sequence $s = (d_i, r_i : i = 1, ..., m)$, where $d_i > 0$ is integer, and r_i is a feasible transfer. The *length* of the schedule is $\sum_{i=1}^{m} d_i$, the scalar d_i is called a *duration*, and m is called the *switching complexity*. A schedule $s = (d_i, r_i : i = 1, ..., m)$ is said to *satisfy* a traffic matrix r if $\sum_{i=1}^{m} d_i r_i = r$.

Note that a lower bound to the minimum length is $L(r) = \max_{e \in E} \lceil \frac{t_r(e)}{c(e)} \rceil$. In section 5.3.2, we provide the outline of an algorithm that finds a schedule of length $L(r)$ that satisfies r. Thus, the schedule has minimum length. In section 5.4, a time complexity analysis of a fast design for this algorithm is given.

For the rest of this chapter we will use the following notation. Suppose $f = (f(x) : x \in X)$, $g = (g(x) : x \in X)$, and X is some set. Then $f \leq$ (resp., $=, \geq$) g is interpreted as $f(x) \leq$ (resp., $=, \geq$) $g(x)$ for all $x \in X$.

5.3.2 The scheduling algorithm

In this section we first define a critical transfer. Informally, for any time slot, a critical transfer is the minimum transfer required to ensure that the lower bound of the schedule length decreases by one unit; a sequence of critical transfers will thus result in an optimum schedule. In the first two lemmas we show the condition a transfer must satisfy in order to be a critical transfer. In the third lemma we show that a critical transfer exists for our system, leading to an algorithm for solving $TreeDTS$.

Def. A feasible transfer g is called *critical* with respect to a traffic matrix h if $g \leq h$ and $L(h-g) = L(h) - 1$.

We also define $b_h = (b_h(e) : e \in E)$ to be a function of traffic matrix h such that $b_h(e) = \max\{0, t_h(e) - (L(h)-1)c(e)\}$.

Lemma 5.1 *Suppose h is a traffic matrix, g is a feasible transfer, and $d > 0$ is an integer such that $h \geq dg$. Then $L(h-dg) = L(h) - d$ if and only if $L(h-dg) \leq L(h) - d$.*

Proof. Since g is a feasible transfer, $t_h(e) - dt_g(e) \geq t_h(e) - dc(e)$. Therefore,

$$\begin{aligned} L(h-dg) &= \max_{e\in E}\lceil \tfrac{t_h(e)-dt_g(e)}{c(e)} \rceil \\ &\geq \max_{e\in E}\lceil \tfrac{t_h(e)}{c(e)} \rceil - d \\ &= L(h) - d. \end{aligned}$$

Since $L(h - dg) \geq L(h) - d$, the lemma is implied. □

Lemma 5.2 *Suppose h is a traffic matrix. A feasible transfer $g \leq h$ is critical with respect to h if and only if $t_g \geq b_h$.*

Proof. By definition, g is critical if and only if $L(h - g) = L(h) - 1$. From Lemma 5.1, we know $L(h - g) = L(h) - 1$ if and only if $L(h - g) \leq L(h) - 1$. The last inequality is equivalent to $\frac{t_{h-g}(e)}{c(e)} \leq L(h) - 1$ for all $e \in E$, because $L(h) - 1$ is integer valued. Next note that $\frac{t_{h-g}(e)}{c(e)} \leq L(h) - 1$ is equivalent to $t_g(e) \geq t_h(e) - (L(h) - 1)c(e)$. Therefore, g is critical with respect to h if and only if for all $e \in E$, $t_g(e) \geq t_h(e) - (L(h) - 1)c(e)$. The lemma is implied since $t_g(e) \geq 0$ for all $e \in E$. □

For the next lemma, and the remainder of this section, we consider another network (V, E^*) (see Fig. 5.5), where the nodes are V, the directed links are $E^* = E \cup E_{RS}$, and $E_{RS} = \{[u, v] : u \in V_R, v \in V_S\}$. (Notice that the links in E_{RS} are oriented from receiver nodes to sender nodes.) A nonnegative vector $f = (f(e) : e \in E^*)$ is called a *flow*. For each $v \in V$, let $I^*(v)$ and $O^*(v)$ be the set of incoming and outgoing links, respectively, for the node v. A flow f is *node-conserved* if for all $v \in V$, $\sum_{e \in I^*(v)} f(e) = \sum_{e \in O^*(v)} f(e)$.

Def. For any traffic matrix h, let the *network link lower bound* $b_h^* = (b_h^*(e) : e \in E^*)$ and *network link capacity* $c_h^* = (c_h^*(e) : e \in E^*)$, such that

$$b_h^*(e) = \begin{cases} b_h(e), & \text{if } e \in E; \\ 0, & \text{if } e \in E_{RS}. \end{cases}$$

$$c_h^*(e) = \begin{cases} c(e), & \text{if } e \in E; \\ \min\{h([v,u]), C(u), C(v)\}, & \text{if } [u,v] = e \in E_{RS}. \end{cases}$$

Note that b_h^* and c_h^* are integer and $c_h^* \geq b_h^*$. Also, for the special case in [24] when "speed-up" is not allowed, we would replace $h([v,u])$ by 1 in the definition of c_h^* above; this change leads to minor modifications in the proofs below which are left to the reader.

Lemma 5.3 *For any traffic matrix h such that $L(h) \geq 1$, there is a critical feasible transfer g.*

Proof. The proof has three steps. We define a flow f and observe that it is node-conserved, show that $b_h^* \leq f \leq c_h^*$, and then use the corresponding integer-valued flow to define a traffic matrix. Let f be a flow such that

$$f(e) = \begin{cases} \frac{h([v,u])}{L(h)}, & \text{if } [u,v] = e \in E_{RS}; \\ \frac{t_h(e)}{L(h)}, & \text{if } e \in E. \end{cases}$$

Then f is node conserved. Next we show that $b_h^* \leq f \leq c_h^*$. Suppose $e = [u,v] \in E_{RS}$. Let e' and e'' be the outgoing link of v and the incoming link of u, respectively. (Notice that here we use the property that leaves of a tree have only one parent). Then $L(h) \geq \max\{\frac{t_h(e')}{c(e')}, \frac{t_h(e'')}{c(e'')}, 1\} \geq \max\{\frac{h([v,u])}{C(v)}, \frac{h([v,u])}{C(u)}, 1\}$; and

$$b_h^*(e) = 0 \leq f(e) = \frac{h([v,u])}{L(h)}$$

$$\leq \min\{C(u), C(v), h([v,u])\} \leq c_h^*(e).$$

Algorithm Tree.
Input: Traffic matrix r.
Output: Schedule $s = (d_i, r_i : i = 1, ..., m)$
that satisfies r.

Let $m = 0$ and $r' = r$.
while $r' \neq 0$ do
Let $m = m + 1$.
Note that $r' = r - \sum_{i=1}^{m-1} d_i r_i$.
Compute a critical feasible transfer
r_m of r'.
Let $d_m > 0$ be the maximum value d such
that $r' \geq dr_m$ and
$L(r' - dr_m) = L(r') - d$.
$r' = r' - dr_m$
end

Table 5.1: The **Tree** algorithm

Suppose $e \in E$. Then $f(e) = \frac{t_h(e)}{L(h)} \leq \frac{L(h)c(e)}{L(h)} = c_h^*(e)$. Also, note that $\frac{t_h(e)}{c(e)} \leq L(h)$; $t_h(e) \leq L(h)c(e)$; $t_h(e)(1 - L(h)) \geq -(L(h) - 1)L(h)c(e)$; $t_h(e) \geq t_h(e)L(h) - (L(h) - 1)L(h)c(e)$; and $\frac{t_h(e)}{L(h)} \geq t_h(e) - (L(h) - 1)c(e)$. Thus, $f(e) \geq b_h(e)$. We are done verifying $b_h^* \leq f \leq c_h^*$.

From the *Integrality Theorem for Flows* (see [122]) we know that since there is a node-conserved flow f such that $b_h^* \leq f \leq c_h^*$, then there is a node-conserved *integer-valued* flow f' such that $b_h^* \leq f' \leq c_h^*$. Let g be a traffic matrix such that $g([u, v]) = f'([v, u])$. Then $t_g(e) = f'(e)$ for all $e \in E$. Note that g is a feasible transfer because $t_g(e) = f'(e) \leq c(e)$; and $g \leq h$ because $g([u, v]) = f'([v, u]) \leq h([u, v])$. Finally, g is critical since $f' \geq b_h^*$ and Lemma 5.2. □

Def. Let $R_i = r - \sum_{j=1}^{i-1} d_j r_j$.

Theorem 5.1 *Algorithm* **Tree** *solves TreeDTS.*

Proof. Lemma 5.3 implies that a critical feasible transfer r_i can always be found for a traffic matrix R_i. Also, the value d_i is greater than zero, because r_i is a critical feasible transfer for R_i. Therefore, the sequence R_1, R_2,... is decreasing in value and the algorithm will eventually terminate with the schedule s. Since $\sum_{i=1}^{m} d_i = \sum_{i=1}^{m}[L(R_i) - L(R_{i+1})]$, the length of s is $L(r)$ and is optimal. □

For the rest of the section we discuss the computation of d_i, which is the maximum value d such that $R_i \geq dr_i$ and $L(R_i - dr_i) = L(R_i) - d$. The inequality $R_i \geq dr_i$ is equivalent to $d \leq \min_{e \in E} \lfloor \frac{t_{R_i}(e)}{t_{r_i}(e)} \rfloor$. From Lemma 5.1, $L(R_i - dr_i) = L(R_i) - d$ is equivalent to $\lceil \frac{t_{R_i}(e) - dt_{r_i}(e)}{c(e)} \rceil \leq L(R_i) - d$ for all $e \in E$. Since $L(R_i) - d$ is integer, the last inequality is equivalent to $\frac{t_{R_i}(e) - dt_{r_i}(e)}{c(e)}$ $\leq L(R_i) - d$, which in turn is equivalent to

$$d \leq \beta_{R_i, r_i}(e)$$

$$= \begin{cases} \frac{L(R_i)c(e) - t_{R_i}(e)}{c(e) - t_{r_i}(e)} & \text{if } c(e) > t_{r_i}(e); \\ \infty, & \text{otherwise.} \end{cases}$$

$$= \begin{cases} \frac{[L(r) - \sum_{j=1}^{i-1} d_j]c(e) - t_{R_i}(e)}{c(e) - t_{r_i}(e)}, & \text{if } c(e) > t_{r_i}(e); \\ \infty, & \text{otherwise.} \end{cases}$$

Therefore, $d_i = \min\{\min_{e \in E} \beta_{R_i, r_i}(e), \min_{e \in E_{SR}} \lfloor \frac{t_{R_i}(e)}{t_{r_i}(e)} \rfloor\}$.

5.4 A time efficient design

In this section we discuss time efficient implementations of Algorithm **Tree** and the time complexity. First we prove a bound on the sum of the link

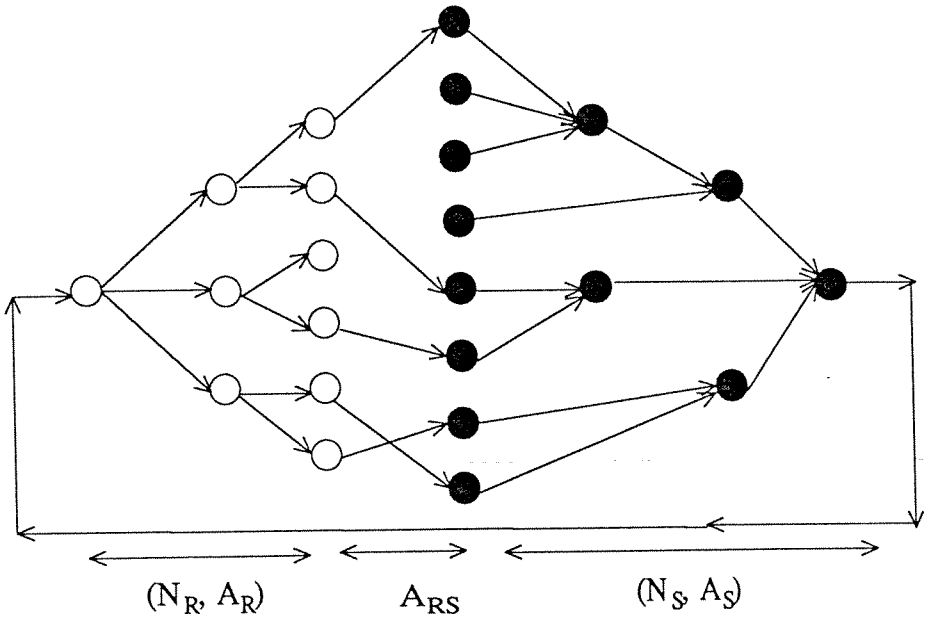

Figure 5.5: Network model $G = (V, E^*)$

capacities in the network (V^*, E^*), and a bound on the switching complexity m of Algorithm **Tree**. These bounds are used in the calculation of the time complexity of our implementation of Algorithm **Tree**.

Lemma 5.4 *Let* $C = \frac{1}{n}\sum_{v \in V_S \cup V_R} C(v)$ *and* $c^* = (c^*(e) : e \in E^*)$ *such that*

$$c^*(e) = \begin{cases} c(e), & \text{if } e \in E; \\ \min\{C(v), C(u)\}, & \text{if } e = [u, v] \in E_{RS}. \end{cases}$$

Then $\sum_{e \in E^*} c^*(e)$ *is* $O(n^2 C)$.

Proof. We know $\sum_{e \in E_{RS}} c^*(e) \leq \sum_{[u,v] \in E_{RS}} [C(u) + C(v)] \leq n \sum_{v \in V_R \cup V_S} C(v)$ $= n^2 C$. Now consider the directed spanning tree $(V' \cup \{x_D\}, E')$, where V' $= V_S \cup V_M$and $E' = E_M \cup \{[x_M, x_D]\}$. Let $c' = (c'(e) : e \in E')$, where $c'(e) =$ $C(v)$ if $v \in V_S$ and e is the outgoing link of v, and if $v \in V_M$ then $\sum_{e \in I(v)} c'(e)$

$= \sum_{e \in O(v)} c'(e)$. Since $\sum_{e \in I(v)} c(e) \geq \sum_{e \in O(v)} c(e)$ for each $v \in V_M$, it follows that $c' \geq c^*$. Let p_v be the length of the path in (V', E') from node $v \in V'$ to node x_D. Then $\sum_{e \in E'} c^*(e) \leq \sum_{e \in E'} c'(e) = \sum_{v \in V_S} p_v C(v)$. Due to the fact that each node $v \in V_M$ has at least two incoming links and (V', E') is a tree, we know $p_u \leq |V_S|$ for any $u \in V'$. Therefore, $\sum_{e \in E'} c^*(e) \leq |V_S| \sum_{v \in V_S} C(v)$ $\leq n \sum_{v \in V_S} C(v)$. Similarly,

$$\sum_{e \in E_D \cup \{[x_M, x_D]\}} c^*(e) \leq n \sum_{v \in V_R} C(v).$$

Thus, $\sum_{e \in E} c^*(e) \leq n \sum_{v \in V_S \cup V_R} C(v)$ and we are done. □

Lemma 5.5 *Suppose h is a traffic matrix such that $L(h) \geq 1$. Let f be an integer, node-conserved flow and g be the traffic matrix such that $g([u,v]) = f([v,u])$. Then g is a critical feasible transfer for h if and only if $b_h^* \leq f \leq c_h^*$.*

Proof. Note that for $e \in E$, $f(e) = t_g(e)$. Next, note that $f \leq c_h^*$ if and only if both g is a feasible transfer and $g \leq h$. Hence, Lemma 5.2 implies that g is a critical feasible transfer for h if and only if $b_h^* \leq f \leq c_h^*$. □

Let f_i be the node-conserved flow such that

$$f_i([u,v]) = \begin{cases} r_i([v,u]), & \text{if } [u,v] \in E_{RS}; \\ t_{r_i}([u,v]), & \text{if } [u,v] \in E. \end{cases}$$

Lemma 5.6 *The switching complexity m of Algorithm* **Tree** *is $O(n^2 C)$.*

Proof. Let the residual traffic at iteration i after a transfer r_i of duration d time slots be $r^d = R_i - d r_i$. From Lemma 5.5, r_i is a critical feasible transfer

for r^d if and only if $b^*_{r^d} \leq f_i \leq c^*_{r^d}$. Then

$$d_i = 1 + \max\{d \geq 0 : b^*_{r^d} \leq f_i \leq c^*_{r^d}\}.$$

Therefore, for each $i < m$, at least one of the two inequalities must hold: $b^*_{R_i} \neq b^*_{R_{i+1}}$ or $c^*_{R_i} \neq c^*_{R_{i+1}}$, i.e., either the lower bound or the upper bound of the flow changes.

Next note that $c^*_{R_i} \geq c^*_{R_{i+1}}$ for $i < m$, because $R_i \geq R_{i+1}$. Since $R_{i+1} = R_i - d_i r_i$, if $e \in E$ then

$$b^*_{R_{i+1}}(e) = \max\{0, t_{R_{i+1}}(e) - (L(R_{i+1}) - 1)c(e)\}$$

$$= \max\{0, t_{R_i}(e) - d_i t_{r_i}(e) - (L(R_i) - d_i - 1)c(e)\}$$

$$= \max\{0, t_{R_i}(e) - (L(R_i) - 1)c(e) + (c(e) - t_{r_i}(e))d_i\}$$

$$\geq b^*_{R_i}(e).$$

If $e \in E_{RS}$ then $b^*_{R_{i+1}}(e) = 0 = b^*_{R_i}(e)$.

Thus, $b^*_{R_1} \leq b^*_{R_2} \leq \ldots \leq b^*_{R_{m+1}}$, $c^*_{R_1} \geq c^*_{R_2} \geq \ldots \geq c^*_{R_{m+1}}$, and $b^*_{R_i} \neq b^*_{R_{i+1}}$ or $c^*_{R_i} \neq c^*_{R_{i+1}}$. This implies $m \leq m + \sum_{e \in E^*}(b^*_{R_1}(e) + c^*_{R_{m+1}}(e)) \leq \sum_{e \in E^*}(b^*_{R_{m+1}}(e) + c^*_{R_1}(e))$, which is $O(n^2 C)$ from Lemma 5.4. □

Theorem 5.2 *Algorithm* **Tree** *can be designed so that it has a time complexity* $O(n^4 C)$.

Proof. Each iteration i of Algorithm **Tree** requires computing r_i, d_i, and updating r'. The implementation of the algorithm we consider will compute r_i by constructing an integer, node-conserved flow f_i such that $b^*_{R_i} \leq f_i \leq c^*_{R_i}$. By Lemma 5.5, the traffic matrix r_i, where $r_i([u,v]) = f_i([v,u])$, is a critical feasible transfer for R_i.

To construct f_{i+1} we consider an implementation that uses f_i, where $f_0 = (0 : e \in E^*)$. Let f'_{i+1} be a flow, that may not be node-conserving, such that

$$f'_{i+1}(e) = \min\{c^*_{R_{i+1}}(e), \max\{f_i(e), b^*_{R_{i+1}}(e)\}\}.$$

Thus, $b^*_{R_{i+1}} \leq f'_{i+1} \leq c^*_{R_{i+1}}$. For each node v, let the surplus flow $\delta_{i+1}(v)$ $= \sum_{e \in I^*(v)} f'_{i+1}(e) - \sum_{e \in O^*(v)} f'_{i+1}(e)$. Let f''_{i+1} be a flow such that $f''_{i+1} = f'_{i+1}$. The flow f''_{i+1} can be modified by the *Feasible Distribution Algorithm* in Rockafellar [122] so that f''_{i+1} becomes integer, node-conserved, and $b^*_{R_{i+1}} \leq f''_{i+1} \leq c^*_{R_{i+1}}$. Then $f_{i+1} = f''_{i+1}$. The Feasible Distribution Algorithm will modify f by applying at most $\frac{1}{2}\sum_{v \in V} |\delta_{i+1}(v)|$ integer flow augmentations, similar to flow augmentations of the Ford-Fulkerson Labeling Algorithm [42]. Each flow augmentation will take $O(n^2)$ time. Therefore, the transformation of f''_{i+1} will take $O(\frac{1}{2}\sum_{v \in V} |\delta_{i+1}(v)|n^2)$ time. Thus, the time to compute $f_1, f_2, ..., f_m$ is $O(Dn^2)$, where $D = \sum_{i=1}^{m} \sum_{v \in V} |\delta_i(v)|$.

Since $|\delta_{i+1}(v)| \leq \sum_{e \in I^*(v) \cup O^*(v)} |f'_{i+1}(e) - f_i(e)|$,

we have

$$D \leq \sum_{i=0}^{m-1} \sum_{v \in V} \sum_{e \in I^*(v) \cup O^*(v)} |f'_{i+1}(e) - f_i(e)|$$

$\leq \sum_{i=0}^{m-1} 2 \sum_{e \in E^*} |f'_{i+1}(e) - f_i(e)|$

$= 2 \sum_{e \in E^*} \sum_{i=0}^{m-1} |f'_{i+1}(e) - f_i(e)|.$

Note that, informally, either $f'_{i+1}(e)$ is increased as the lower bound increases to $b^*_{R_{i+1}}(e)$, or is decreased as the upper bound decreases, i.e.,

$$
\begin{aligned}
&|f'_{i+1}(e) - f_i(e)| \\
&= \max\{b^*_{R_{i+1}}(e) - f_i(e), 0\} \\
&\quad + \max\{f_i(e) - c^*_{R_{i+1}}(e), 0\} \\
&\leq b^*_{R_{i+1}}(e) - b^*_{R_i}(e) + c^*_{R_i}(e) - c^*_{R_{i+1}}(e).
\end{aligned}
$$

Therefore, $D \leq \sum_{e \in E^*} c^*(e)$, and applying Lemma 5.4, D is $O(n^2C)$. This implies that the time it takes to compute $f_1, f_2, ..., f_{m+1}$ is $O(n^4C)$.

For each iteration i of Algorithm **Tree**, the time to compute d_i, $b^*_{R_i}$, and $c^*_{R_i}$ and to update r' is $O(n^2)$. By Lemma 5.6, the number of iterations is $O(n^2C)$. We can conclude that the time complexity of the algorithm is $O(n^4C)$. □

5.5 Experimental evaluation

In this section we describe the results of an experimental evaluation of the performance of the **Tree** algorithm. since the algorithm always produces an optimal schedule, the key question is the time that it takes to produce that schedule.

We implemented the **Tree** algorithm as a C program, generally along the lines we have discussed in this chapter. The main difficulty was in implementing the routines to find augmenting paths to modify a flow so as to make it node-conserved. The shortest augmenting path subroutines were implemented using the algorithms sketched in Gibbons [58] as a guide.

The program implementing **Tree** was evaluated by measuring the CPU time it took to execute when presented with uniformly randomly generated graphs as inputs. Random graphs were generated for selected combinations of the input parameters using a pseudo-random number generator [94]. The programs were executed on a Sun Sparc 2 workstation running the SunOS^{TM} Release 4.1.1 operating system after being compiled using the Sun Microsystems C compiler (bundled with SunOS Release 4.1.1), with Level 4 optimization enabled ("-O4" option). The data structures for the program fit in the 32 MB main memory of the system, and so the program does not perform any I/O in order to execute, except to read the input graph and print results.

Tree expt. 1. Effect of varying number of senders. The first experiment estimated the performance of **Tree** for architecture graphs in which both the multiplexer tree and the demultiplexer tree are complete balanced binary trees with unit capacities for all links, and all edges in the resource graph have unit weights. Thus the average capacity of the user links in the architecture graph is $C = 1$, the number of senders and receivers is the same, and if there are S senders there are $(S - 1)$ multiplexers, each with two incoming links and one outgoing link. Clearly S is constrained to be a power of 2, and once it is chosen the structure of the architecture graph, and the number of nodes in the resource graph, is completely fixed.

The only random variable remaining in the architecture graph is the capacity of the link from the root of the multiplexer tree to the root of the demultiplexer tree. The capacity of this link is set to be an integer drawn randomly from the interval $[2, S/2]$. The edges of the resource graph (i.e., the entries of the traffic matrix) were generated by a pseudo-random number generator [94]. Each edge exists with probability 0.5; if it exists, it carries unit traffic.

For each value of S, one hundred random input instances, or *batches*, were generated as described above, and the CPU time taken by **Tree** to generate a schedule for the entire one hundred batches, as reported by the C Shell "time" command, was recorded. The "time" command used has a resolution of 20 ms. Since all (except one) of the measurements are of times greater than 4 seconds, and many are of times of hundreds of seconds, this resolution is sufficient. The mean for each set of 100 measurements was calculated, plotted and curve-fitted as described in section 3.7.

In Fig. 5.6 the mean CPU time per batch taken by **Tree** as the number of senders (or receivers) S is increased, is shown. The curve-fit shown in the figure corresponds to the following equation and correlation coefficients:

$$Tree(n) = 2.78 \times 10^{-5}\ n^3 - 9.26 \times 10^{-4}\ n^2 - .055,$$

$$R3^2 = .99,\ R2^2 = .98,\ R1^2 = .89,\ R0^2 = 1.0$$

For the sake of completeness, some data on 'internal' measures of performance of **Tree** are summarized in Table 5.2. These are the average number

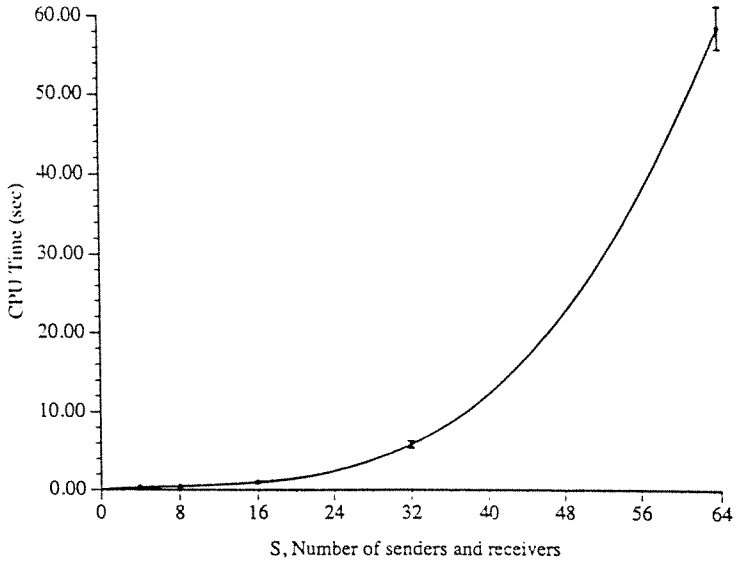

Figure 5.6: CPU time versus number of senders or receivers for unit-length transfers and complete binary tree architectures

of unit-length transfers to be scheduled per batch, the average optimal schedule length per batch in time slots, the average number of augmenting paths calculated in order to find the optimal schedule, and the average CPU time per batch.

Tree expt. 2. Effect of varying transfer lengths. In the second experiment, the architecture graph of the input instances to **Tree** is kept fixed and only the lengths of transfers are varied. The architecture graph is fixed to be a balanced binary tree with 64 senders and unit capacity links. Each edge in the resource graph exists with probability 0.5; it is labeled with the length of the transfer by a random number drawn from the interval $[1, K]$, where K is the maximum transfer length and is varied from 2 to 1024. For each value of K, one hundred batches are generated randomly as described above, and the time for calculating the optimal schedule for the entire one hundred

Senders, S	Transfers	Makespan	Paths	CPU Time (sec)
8	7.99	5.0	7.46	.014
16	31.28	17.77	29.96	.050
32	127.94	68.34	123.97	.365
64	515.09	266.61	501.27	4.362

Table 5.2: Behavior of **Tree** as number of senders is varied for balanced binary trees of unit capacities and unit-length transfers

Maximum Transfer Length, K	Transfers	Traffic	Make-span	Switchings	Paths	CPU Time (sec)
1	515.09	515.09	266.61	266.61	501.27	4.362
2	515.09	771.88	401.76	348.69	505.78	5.242
16	515.09	4377.62	2290.68	486.37	513.54	6.709
128	515.09	33217	17402	511.35	514.86	6.982
512	515.09	132092	69207	514.21	515.06	6.999
1024	515.09	263929	138282	514.53	515.06	7.017

Table 5.3: Behavior of **Tree** as maximum transfer length is varied for balanced binary trees of unit capacities and 64 senders

batches is measured using the "time" command, as for Tree expt. 1. Figure 5.7 shows the mean CPU time taken to calculate the schedule, per batch, as K is varied.

Table 5.3 shows 'internal' measures of the behavior of **Tree** as K is varied, averaged per batch. Since transfers are no longer of unit length, the total traffic per batch does not equal the number of transfers (unlike Table 5.2), and the makespan does not equal the switching complexity. Thus these quantities are displayed separately.

Tree expt. 3. Effect of varying link capacities. In the third experiment,

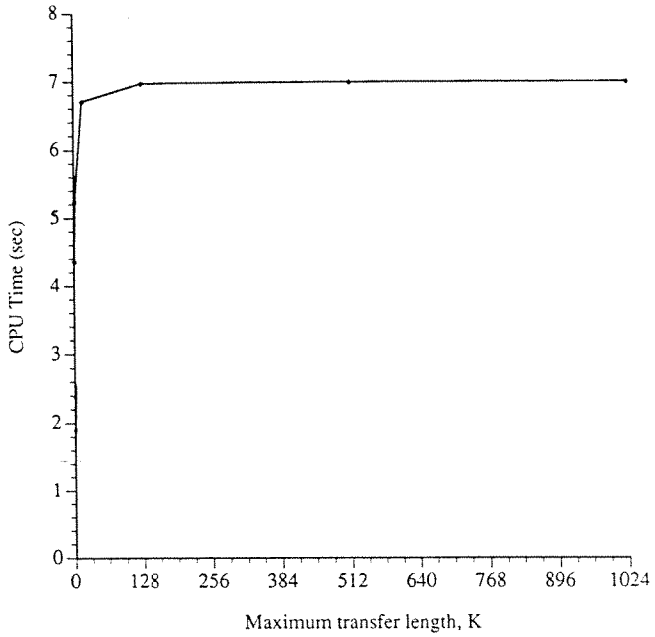

Figure 5.7: CPU time versus maximum transfer length for complete binary tree architectures with 64 senders

balanced binary trees were again considered, but the capacity of the user links, C, was varied as described below. The maximum transfer length was set at $K = 2$.

Recall that the architecture graph for $TreeDTS$ contains sending users connected to a bank of multiplexers. In order to keep the structure of the graph 'sensible', it was assumed that the sum of the incoming link capacities to a multiplexer is at least equal to the outgoing link capacity (and the analogous assumption for demultiplexers). When $C = 1$, and the tree is not degenerate (i.e., the branching factor is at least two), this assumption is easily captured by generating input architecture graphs in which all links have unit capacity. For $C > 1$, we capture this assumption by setting the capacity of all user links to exactly C. The capacity of non-user links in the architecture graph is then generated as an integer drawn randomly from the interval [1, C'], where

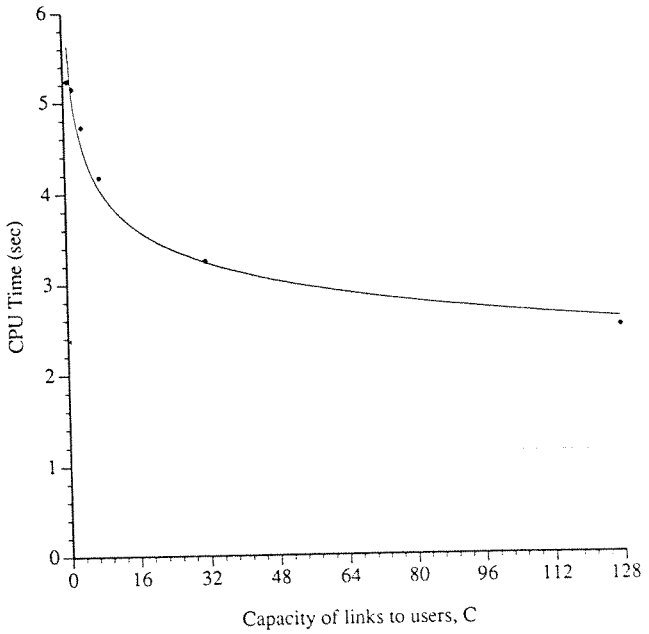

Figure 5.8: CPU time versus user link capacity for complete binary tree architectures with 64 senders

C' is the sum of the incomping links to a multiplexer (or outgoing links at a demultiplexer).

Figure 5.8 shows the mean CPU time per batch for generating an optimal schedule using Tree for 100 batches of randomly generated inputs as C is varied from 1 to 128. The curve is described by:

$$Tree(C) = 5.64C^{-0.16}, \qquad R^2 = .98$$

The internal performance measures of the algorithm are summarized in Table 5.4.

It is interesting to note that the execution time of the algorithm decreases as C is increased, if the architecture and resource graph (traffic) inputs are

User Link Capacity, C	Transfers	Traffic	Make-span	Switchings	Paths	CPU Time (sec)
1	515.09	515.09	266.61	266.61	501.27	4.362
2	515.09	771.88	383.23	335.20	505.88	5.172
4	515.09	771.88	337.21	295.48	506.24	4.729
8	515.09	771.88	284.32	249.04	512.69	4.181
32	515.09	771.88	172.73	151.0	541.20	3.251
128	515.09	771.88	78.56	73.42	573.88	2.498

Table 5.4: Behavior of **Tree** as user link capacity is varied for balanced binary trees of 64 senders

kept the same. Informally, the explanation for this is straightforward: as C is increased, more traffic can be transferred per time slot, so that the switching complexity (number of feasible transfers that need to be calculated) decreases. The multiplicative factor of C in the theoretical asymptotic time complexity arises from assuming that in the worst case, at each time slot, the capacity or lower bound of any link changes by at most one unit; then the number of times that feasible transfers have to be calculated is proportional to the maximum difference between capacity and lower bound for any link, which in turn is proportional to C. In order to bring the theoretical and experimental results in closer intuitive agreement, another method should be used to bound the switching complexity of the algorithm, i.e., the theoretical analysis should be sharpened.

5.6 Discussion

5.6.1 Previous related work

The previous work for scheduling data transfers in tree architectures has all been done for special cases of the $TreeDTS$ problem. In addition, it has all been done in the context of specific application domains. The result is that the previous research has been reported in a wide variety of journals and conferences, using specialized notation and jargon, and with no cross-referencing of research across application areas. In the following we summarize some of this research using our scheduling model.

The ground-breaking work on data transfer scheduling was done by Coffman, Garey, Johnson and LaPaugh, in their 1985 paper which focused on the problem of non-preemptive file transfers in a distributed network [27]. The problem they address differs from $TreeDTS$ in that $Preempt = false$, and general architecture graphs were considered. The paper is remarkable in defining a new and interesting class of problems and presenting a large number of results, including NP-completeness of various classes of problems, approximation algorithms for these problems with performance guarantees, polynomial-time algorithms for special cases, and distributed algorithms for some situations. The suggestions for future work mention the realistic situation of file transfers where intermediate network nodes may be used to forward files if the sender and receiver have no direct link. These and other variations of the problems mentioned in the paper have subsequently been addressed by several researchers (e.g., see [23, 142, 101] and references therein), and continue to receive attention.

Our work differs from that of Coffman et al [27] and their successors in that we are considering the situation where preemption is allowed. This assumption not only makes the scheduling problem easier, but is well-justified for many applications. For instance, in the case of parallel I/O, data is typically read in fixed-size blocks from magnetic media, allowing preemption between block boundaries. Similarly, in the case of satellite TDMA switching, data is transferred in fixed-size packets in order to time-share the medium using time-division multiplexing, and communications protocols often assume packetization of data in order to operate correctly. In the rest of this section, we consider preemptive scheduling only.

Eng and Acamopra [39] consider a special case of $TreeDTS$ for the satellite switching application. They place the following additional restrictions on the architecture graph:

1. The hierarchical switching system has three stages, i.e., AG is a tree with exactly four levels (a dummy root, and two subtrees of three levels each), and
2. arcs connected to leaves (users) have a capacity of exactly 1 packet per second, i.e., $C = 1$.

Eng and Acampora [39] provide necessary and sufficient conditions for the existence of an optimal schedule. Bonnucelli [11] and Liew [96] provide exact optimal scheduling algorithms of time complexity $O(n^5)$ under these conditions. Informally, the basic approach is, at every time slot, to identify critical transfers - those whose remaining traffic demand equals the optimal makespan

calculated using Eng and Acampora's conditions. At every time slot as many packets as possible from these critical transfers are transferred - the number that can be sent is calculated using a max-flow algorithm. The resulting schedule will have the optimal makespan. We note that our algorithm will solve this restricted problem in time $O(n^4)$.

Choi and Hakimi [24] study a special case of $TreeDTS$ for the file transfer application. They place the following constraints on the architecture graph:

1. AG is a tree with exactly three levels, and
2. a) arcs of AG connected to leaves (users) have an arbitrary positive integer capacity equal to the number of ports, (this case is called *speed-up*) or
   b) arcs of AG connected to leaves (users) have capacity exactly 1 and (this case is called no speed-up)

In addition, preemption is allowed arbitrarily, i.e., packets may be of variable length. We shall show in a later chapter that this extension can be handled by a simple modification of $TreeDTS$.

Choi and Hakimi's problem generalizes that of Bonnucelli [11] and Liew [96] in allowing non-unit capacities and allowing variable-length packets, but is restricted in allowing only three levels in the AG. The term speed-up implies that multiple ports may be used to expedite the data transfer between a given sender-receiver pair. Thus allowing a vertex v to engage in upto $\alpha(v)$ transfers means that a transfer of length w between vertices u and v only requires time $w/(\min \alpha(u), \alpha(v)))$.

Choi and Hakimi's method is interesting because it illustrates an alternative approach to the problem. They essentially use a generalized edge-coloring method in order to obtain optimal schedules. Their problem is a generalization of the *SimpleDTS* and *DTS* problems that we have solved using standard edge-colorings of bipartite graphs, but is a restriction of *TreeDTS*, for which we used more powerful network flow methods. Further, they handle the case where at most $k \leq n$ transfers may take place at any given time by using a procedure similar to the k-filling procedure we use for *DTS*. Thus not only their problem, but their solution method falls at a point in between the approaches that we have taken to solve the *DTS* and *TreeDTS* data transfer scheduling problems.

Under speed-up, Choi and Hakimi's algorithm runs in time $O(C_{sum}m^2)$, where C_{sum} is the sum of the link capacities at the user nodes and m is the number of data transfers, i.e., it runs in time $O(Cn^5)$. Without speed-up the algorithm runs in time $O(C_{sum}m^2 + C_{sum}^2 m)$, i.e., $O(Cn^5 + C^2n^4)$.

In the same paper, Choi and Hakimi [24] also consider the situation where a non-zero *switching time* is required in order to change from one switch configuration to another. They show that even a simple version of this problem is NP-complete, and design approximation algorithms.

Our formulation of the problem allows modeling of speed-up since all arcs connected to leaves have capacities equal to the number of ports. For the case without speed-up we modify the network model so that arcs representing the data transfers themselves (arcs from receiving users to sending users)

have capacity 1. In either case our algorithm generalizes and provides a faster algorithm for the problem than the optimal algorithms of Choi and Hakimi [24], provided that preemption is allowed only at fixed packet length boundaries. As already mentioned, in a later chapter we will show that even for the latter case, the **Tree** optimal algorithm can be extended to generalize and have better time complexity than Choi and Hakimi's optimal algorithms.

The data transfer problem has continued to receive attention for the satellite switching system application. Chalasani and Varma [20] present an algorithm for the restricted system studied by Eng and Acampora [39, 11, 96] that takes time $O(n^2 \min(L, n^2) \min(k, n^{.5}))$, where L is the schedule length and k is the capacity of the central switch in the three-stage network. Thus their algorithm takes time $O(n^{4.5})$.

Chalasani and Varma [136] have also presented an algorithm whose basic idea is similar in spirit to the approach we have used to design **Tree**: instead of recalculating flows from scratch at each step of the algorithm, calculate only a modification to the flow. However, their algorithm again only applies to the three-stage unit-capacity network specified by Eng and Acampora [39], and runs in time $O(n^2 + cn)$, where c is the number of traffic units to be reassigned in the traffic matrix, i.e., it runs in time $O(Ln^2)$, where L is the schedule length. There are two interesting points in their approach, however. The first is that they use a procedure similar to k-filling to convert the problem for a three-stage hierarchical architecture to that for a single switch (the architecture graph of DTS). The second is that they use an algorithm originally developed for routing traffic in a switching network to perform scheduling.

It is clear that the **Tree** algorithm solves more general cases of the optimal data transfer scheduling problem than those considered in [39, 11, 96, 20, 136]. It also solves more general cases of the problem than the polynomial-time solvable problems studied in [24]. In general, for arbitrary traffic matrices (and hence schedule lengths that may be greater than $O(n^2)$), **Tree** also performs better than previous algorithms [39, 24, 11, 96, 20, 136].

5.6.2 Conclusions and Further Work

We have defined the problem of data transfer scheduling in tree-structured architectures in our model, and shown that it generalizes previous work done in three different application areas: parallel I/O [132], satellite switching systems [39, 11, 96], and file transfers in computer networks [24]. The generalization comes in two forms: in allowing arbitrary tree topologies in the architecture, and in allowing arbitrary integer capacities for the communications links. We have developed an algorithm whose time complexity is $O(Cn^4)$, where C is the average capacity of the links connected to senders and receivers, and n is the number of senders and receivers. In general, this algorithm is faster than previous algorithms [24, 11, 96, 20, 136].

Our study has also been unique in that we have actually implemented the scheduling algorithm and investigated its behavior experimentally. For the case $C = 1$, unit transfer lengths, and complete binary tree architecture graphs, we have found that the CPU execution time of the algorithm grows, on average over randomly generated input instances, as $O(n^3)$ in our experiments, rather than the $O(n^4)$ worst-case theoretical behavior. As expected, the execution

time of the algorithm is experimentally observed to be relatively insensitive to transfer lengths. However, we observed that, for a fixed architecture and traffic requirement, the execution time dropped as the user link capacity was increased. While we ofer an informal explanation for this result, it indicates that the theoretical analysis can be sharpened.

An obvious direction for extensions to this work is consider the use of heuristics in place of the optimal algorithm. We shall consider one such heuristic in Chapter 6. We have also considered the situation where data transfers can be preempted at arbitrary points, i.e., data packets can be of arbitrary length. This result is discussed in a later chapter.

For future work, we suggest two related questions. The first is whether the architecture graph can be generalized from a tree topology to an architecture with a higher connectivity. An example would be a hypercube network. Such an extension to our scheduling algorithms would be of significant practical interest. For instance, Ghosh and Agarwal [57] have proposed a hypercube network for I/O, in addition to the existing hypercube network for interprocessor communication, to overcome the parallel I/O bottleneck in hypercube multiprocessor architectures such as the Intel iPSC/2 and iPSC/860. An extension to higher-connectivity architectures would also be of theoretical interest in order to investigate whether polynomial-time or pseudo-polynomial time algorithms could be then be developed. So the second question that we suggest for future work is to determine the smallest set of extensions to the tree architecture defined in $TreeDTS$ which still allow polynomial-time algorithms to be developed.

Chapter 6

A Fast Heuristic for Hierarchical Architectures

In the previous chapter we discussed an optimal algorithm for scheduling data transfers in tree structured architectures, and a theoretical as well as experimental evaluation of its performance. This optimal algorithm, **Tree**, thus applies to the *TreeDTS* scheduling problem, and has a time complexity of $O(Cn^4)$, where n is the number of users (or leaves in the architecture graph), and C is the average capacity of the links connected to the users. The experimental evaluation showed that, for the situations studied, the average execution time of the algorithm varied as n^3 rather than n^4 for random input graphs. While the theoretical worst-case time complexity is an improvement over previous algorithms in this area, and the experimentally measured average-case behavior gives us better performance than the worst-case behavior, the algorithm may still not be fast enough for some applications. Since the algorithm will be executed repeatedly in any realistic batch scheduling situation, any gain in speed will help improve overall system performance. This motivates the search for faster approximation algorithms to solve *TreeDTS*.

In this chapter we present an approximation algorithm based upon a simple greedy heuristic. In sec. 6.1 we describe the approximation algorithm, and

in sec. 6.2 we describe an experimental evaluation of its performance, as well as a comparison of its performance with that of **Tree**. We end with a brief discussion.

6.1 A greedy heuristic

The basic idea of the greedy heuristic for $TreeDTS$ is quite simple, and similar in spirit to the greedy heuristics for $SimpleDTS$ and DTS discussed in Chapter 4. For each time slot, a set of transfers between sender-receiver pairs is chosen such that it is 'feasible', i.e., the resulting traffic on any link in the network does not exceed the capacity of that link. The "greedy" aspect of this approach lies in that as large a set as possible is chosen for each time slot; the "heuristic" aspect lies in that for each time slot, the transfers are examined once, in some order, and a transfer is added to the feasible set only if the resulting set will not violate the link capacity constraint. An algorithm based on this idea is described below.

Algorithm Greedy Tree Heuristic

Input: An instance of $TreeDTS = (PG, AG, RG, f, Preempt)$, where the bipartite resource graph is denoted $RG = (A, B, E)$.

Output: A schedule satisfying $TreeDTS$, represented as a set of feasible transfers for each time slot.

1. Assign some order $F = \langle e_1, e_2, ..., e_m \rangle$ to the edges of E
2. i := 0

```
3. while F ≠ {} {
4.     E' := {}          /* Feasible transfers for this time slot */
5.     for each e read in sequence from F {
6.         if the traffic due to {e} ∪ E' for any link in AG does not
           exceed that link's capacity {
7.             add e to E'
8.             remove e from E and F
9.         }
10.    }
11.    i := i + 1
12.    Assign an order F to the edges of E
13. }
```

Clearly, the performance of the heuristic will depend upon the ordering F of the edges in the resource graph. The bulk of the execution time of the algorithm will be spent in sorting the edges to obtain F, and in checking for each transfer and each time slot that the edge set E' represents a set of feasible transfers.

The heuristic that we have investigated simply orders the edges in E by the order in which they are presented to the scheduler, i.e., essentially a random ordering. Thus obtaining F from E takes no time. We call the resulting algorithm the *Greedy Random Assignment* (**GRA**) algorithm.

6.2 Experimental evaluation of the greedy heuristic

The **GRA** algorithm was implemented and evaluated experimentally in a manner similar to that described in 5.5 for **Tree**. In fact, many of the underlying routines in the implementation used were the same, as was the 'driver' program for parsing inputs and keeping statistics. Also, **GRA** was evaluated for the same set of input parameters as described for **Tree**, and using the same random number generator, seeds, and number of input graphs per experiment. Specifically, the three experiments described as **Tree expt. 1 - 3** in sec. 5.5 were repeated, with the **Tree** algorithm replaced by the **GRA** algorithm; the resulting experiments are called **GRA expt. 1 - 3**. There is one difference in the results obtained: since **GRA** is an approximation algorithm, there are two measures of peformance: the makespan of the schedule calculated, and the amount of time to needed to calculate the schedule. In the following, the figures presented for **Tree** in sec. 5.5 are shown again to allow easy comparison with those for **GRA**.

GRA expt. 1. Effect of varying nuumber of senders. The performance of **GRA** is compared with that of **Tree** for this experiment in Figures 6.1 and 6.2. From Fig. 6.1 we see that as the number of sending (or receiving) users grows, the savings in execution time obtained by using the heuristic increases. The curve fit is given by the equation:

$$GRA(n) = 7.89 \times 10^{-4}\, n^2 - 2.35 \times 10^{-2}\, n + 0.173,$$

$$R2^2 = .98,\ R1^2 = .90,\ R0^2 = .99$$

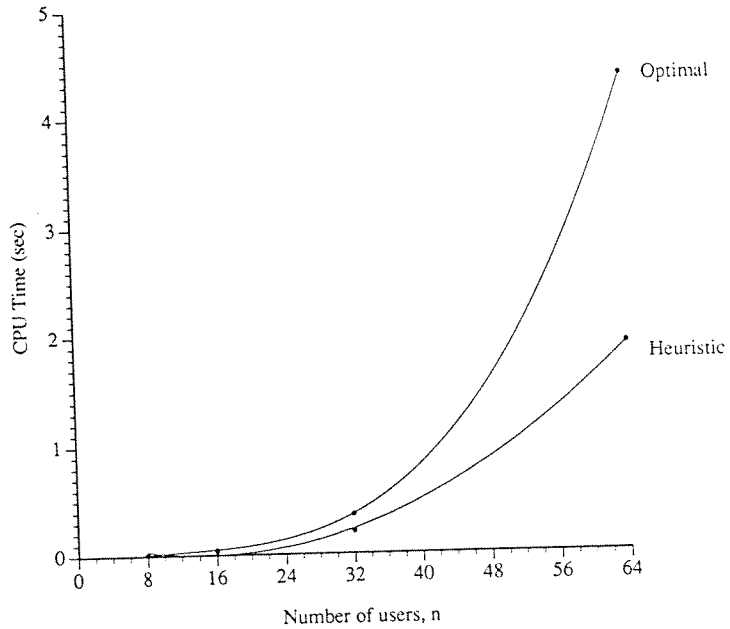

Figure 6.1: CPU time for **GRA** and **Tree** versus number of senders or receivers for unit-length transfers and complete binary tree architectures

Thus the increase in execution time of **GRA** for this experiment as the number of senders is increased is $O(n^2)$, as opposed to $O(n^3)$ for **Tree**. This improvement obviously comes for a price: the schedule obtained by **GRA** is, in general, not of minimum length. In Fig. 6.2 we consider the percentage increase in the schedule length obtained by **GRA**. Recall that for each value of n, one hundred batches were generated at random as inputs. For each batch, the percentage increase in schedule length was calculated. The figure shows the maximum and average values of the percentage increase over the one hundred batches. While the maximum percentage increase observed was almost 35%, the average penalty was only about 5% or less.

GRA expt. 2. Effect of varying transfer lengths. The execution time of **GRA** as the transfer length is varied is compared with that of **Tree** in Figure 6.3, and the percentage penalty paid in terms of schedule length is shown in

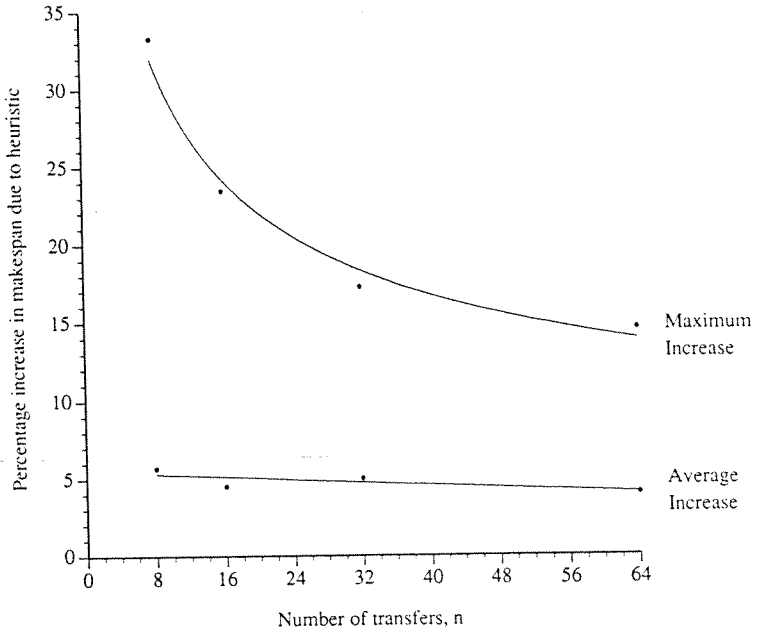

Figure 6.2: Maximum and average penalty paid for using the **GRA** heuristic instead of the **Tree** algorithm, versus number of senders or receivers for unit-length transfers and complete binary tree architectures

Fig. 6.4.

GRA expt. 3. Effect of varying capacities. The execution time of **GRA** as the user link capacities are varied is compared with that of **Tree** in Figure 6.5, and the percentage penalty paid in terms of schedule length is shown in Fig. 6.6. The curve-fit for the variation in execution time is given by the equation:

$$GRA(C) = 2.79C^{-0.25}, \qquad R^2 = .95$$

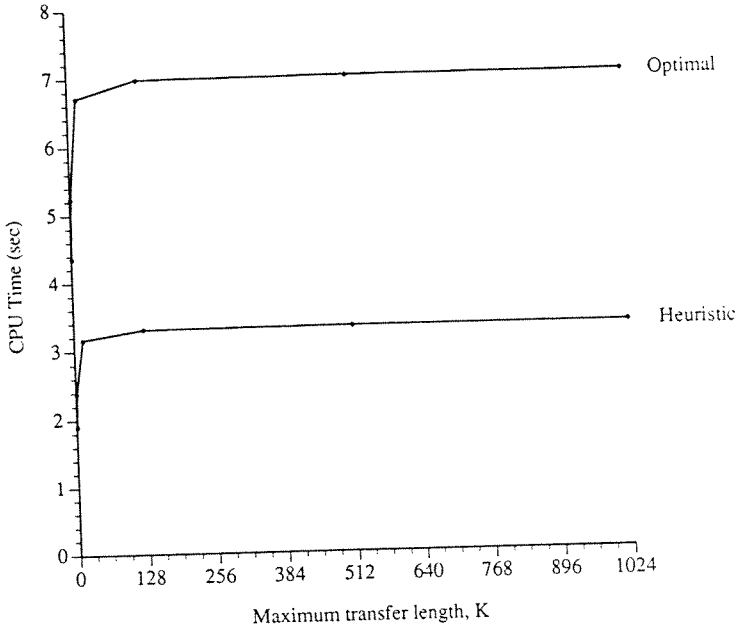

Figure 6.3: CPU time versus maximum transfer length for **GRA** and **Tree** for complete binary tree architectures with 64 senders

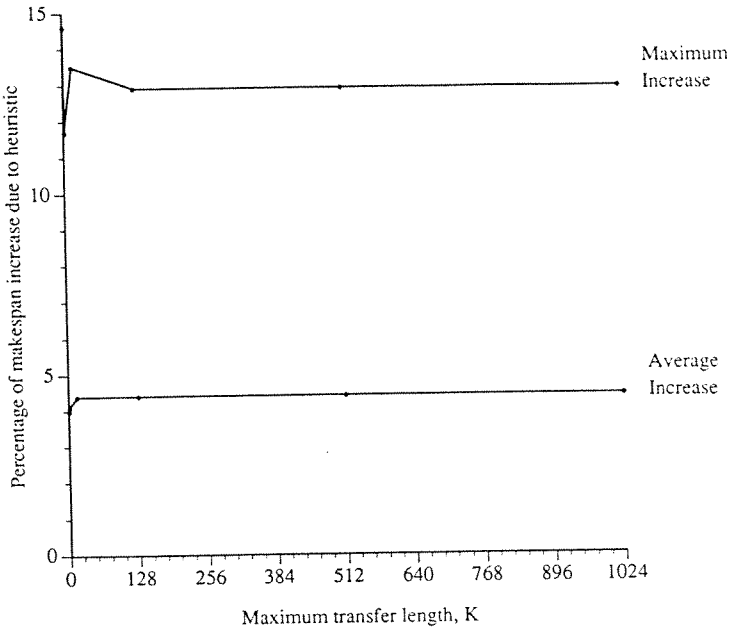

Figure 6.4: Penalty paid for using **GRA** versus maximum transfer length for complete binary tree architectures with 64 senders

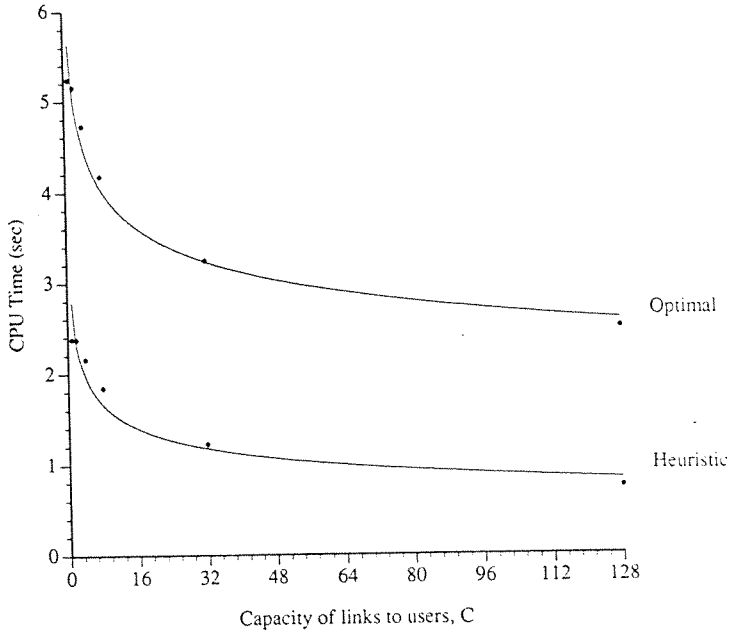

Figure 6.5: CPU time versus user link capacity for **GRA** and **Tree** for complete binary tree architectures with 64 senders

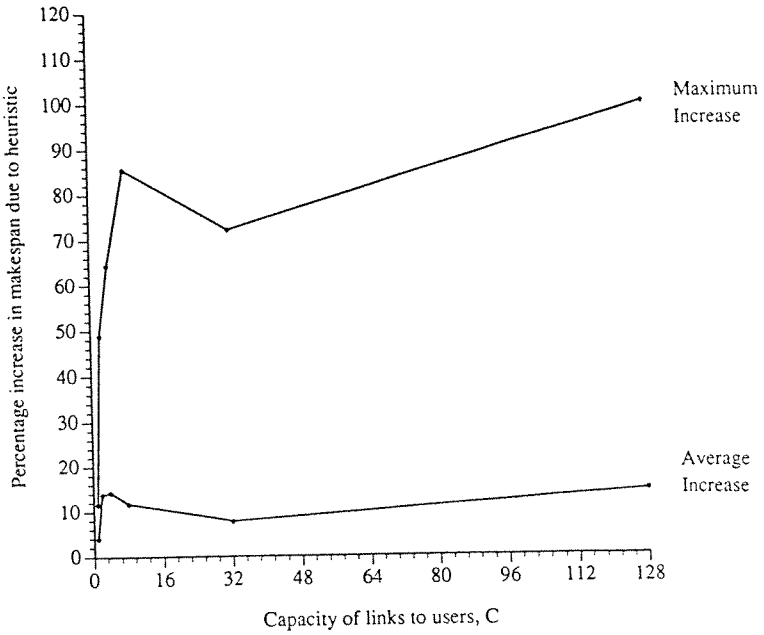

Figure 6.6: Penalty paid for using **GRA** versus user link capacity for complete binary tree architectures with 64 senders

6.3 Discussion

The experiments described in the previous section show that as the number of users is increased, the **GRA** algorithm can provide substantial savings in average execution time over **Tree** (upto 50%, or an asymptotic factor of n improvement) for a relatively small average penalty (about 5% increase in schedule length). This behavior is almost independent of the lengths of the transfers involved. The insensitivity to transfer lengths is as expected, since although longer transfer lengths will lead to longer schedules, they will in general not increase the switching complexity, or number of feasible transfers that are calculated; each feasible transfer will simply be re-used for a greater number of time slots.

The running time of **GRA** follows that of **Tree** as the user link capacities are incresed. The discussion in sec. 5.6 applies to **GRA** also. It is interesting to note that as user link capacities are increased, the maximum penalty, in terms of schedule length, for using **GRA** increases quite sharply, while the average penalty remains in the 10-15% range. Clearly as link capacities increase the optimal algorithm has greater opportunities for maximizing the number of parallel data transfers at every time slot, while the heuristic does not backtrack to try to take advantage of these opportunities once a set of feasible transfers has been found. Further study is needed to fully understand the differences between the average and maximum penalties, and the effect of increasing the user link capacities even further.

Taking a broader view, the results of this chapter constitute one exploration of a design space for constructing heuristics to solve $TreeDTS$. It is possible to

design a different heuristic, for instance simply by changing the ordering that is applied to the edges of the resource graph in the Greedy Tree Algorithm. As a concrete example, consider the following promising heuristic: sort the edges by their congestion, i.e., calculate for each edge the ratio of the transfer length for that edge to the minimum of the capacities of its end users, and examine those edges with the highest congestion ratios first. Call this heuristic the Maximum Congestion First, or **MCF** heuristic. The **MCF** heuristic is myopic in the same sense that **GRA** is: only part of the input instance information is examined. **MCF** is also simplistic in the same sense that **GRA** is: no backtracking is done to improve on the initial choices made by the algorithm. However, **MCF** would give rise to different, probably better, performance in terms of schedule length, while paying a greater penalty in terms of execution time, than **GRA**. An experimental evaluation of **MCF**, very similar in style to that described in this chapter for **GRA**, could then be carried out to test and quantify its behavior.

This example shows a general characteristic of the process of designing heuristics to solve optimization problems. There are two classes of design parameters: the amount of the (input) state space that is examined, and the complexity of the function applied to it. In the case of the **GRA** and **MCF**, both these parameters are changed: **MCF** examines more of the state space, as well as applies a slightly more complex function to it, than **GRA**. One can envision a range of heuristics that can be systematically designed, each appropriate for different input parameters and applications, as the design parameters are systematically varied.

Chapter 7

Scheduling in Extended Hierarchical Architectures

In this chapter we extend the range of problems that can be solved still further. The first extension solves a generalization of the problem of scheduling in tree architectures that was studied in Chapter 5. In the following section we consider tree architectures in which some transfers do not traverse the root of the tree, i.e., both local and remote transfers are permitted. When this possibility is specified in our model, the architecture graph is no longer a tree. This problem has applications both for parallel I/O and satellite communications scheduling. We obtain an approximation algorithm that solves more general cases of the problem than the best previous heuristic, has the same performance guarantee, but a better time complexity.

The second extension in this chapter considers tree architectures in which preemptions may take place arbitrarily, and not only at pre-specified integer boundaries. This problem is applicable to data transfers in systems with continuous media, such as those in current and proposed multimedia systems. We show that the **Tree** algorithm can be modified slightly to solve this problem. The last extension we mention is that of tree-structured architectures in

which the users communicate via transceivers. This problem has applications in packet radio networks. Sasaki [125] has proved that an approximation algorithm can be designed for this problem. The **Tree** algorithm modified to handle arbitrary preemptions is used as a subroutine in the solution.

7.1 Systems with local and remote data transfers

In this section we consider an extension to the $TreeDTS$ problem in which the architecture graph permits both local and remote transfers i.e., transfers that pass through the root of the tree architecture (remote transfers), as well as transfers that only traverse the root of a subtree (local transfers). This is an important extension as it occurs frequently both for the parallel I/O application as well as the communications switching application. In the case of the parallel I/O application it occurs in shared-bus systems such as the Sequent [100] and potentially also in systems such as Hector [138] and interconnection networks such as KYKLOS [102]. In the case of communications networks it occurs for intersatellite transfers [7, 53, 54].

We first define the problem formally in our model, and discuss how it arises in the parallel I/O and intersatellite communications application. We then show how the graph-theoretic nature of our specification facilitates the systematic *decomposition* of the problem into a number of subproblems, allowing us to obtain a heuristic solution. This solution also applies to a special case of the intersatellite communication problem studied earlier by Bertossi et al [7], and provides the same upper bound on makespan but better time complexity than the best heuristic available previously.

7.1.1 Specification of the problem

A formal specification of the problem in the model is as follows. See Fig. 7.1 for an example.

LocalRemoteDTS = (PG, AG, RG, f, Preempt)

where the 5-tuple is defined as follows.

$PG = (T, Ep, Lp)$ has $|T| = m$ tasks, of length $Lp(t) \in N$ for all $t \in T$, and $|Ep| = 0$, i.e., no precedence constraints.

$AG = (R, Ea, La)$ has $|R| = n + 3b$ vertices, with n vertices of type $SUSER$ and $RUSER$ b vertices each of type MUX and $DMUX$, and b vertices of type $NULL$. Each $NULL$ vertex is the root of a three-level tree which can be specified exactly as the architecture graph of *TreeDTS*, with N leaves of each type $SUSER$ and $RUSER$. (Hence $n = 2bN$). One of the $NULL$ vertices, called the *system vertex*, has an incoming arc from the root of every other $DMUX$ subtree to the root of its MUX subtree and an outgoing arc from the root of its $DMUX$ subtree to the root of every other MUX subtree. (These arcs are called *inter-bus links*, a name suggested by the parallel I/O application). The capacity is $La(e) = 1$ for all $e \in Ea$ except those arcs incident on the $NULL$ vertices; for the latter arcs $La(e) \geq 1$ and, for the parallel I/O application, represents the bus capacity.

$RG = (R, Er, Lr)$ is a bipartite graph, where Er is a set of arcs from $SUSER$ to $RUSER$ vertices only, leaving the assignment of other resources implicit,

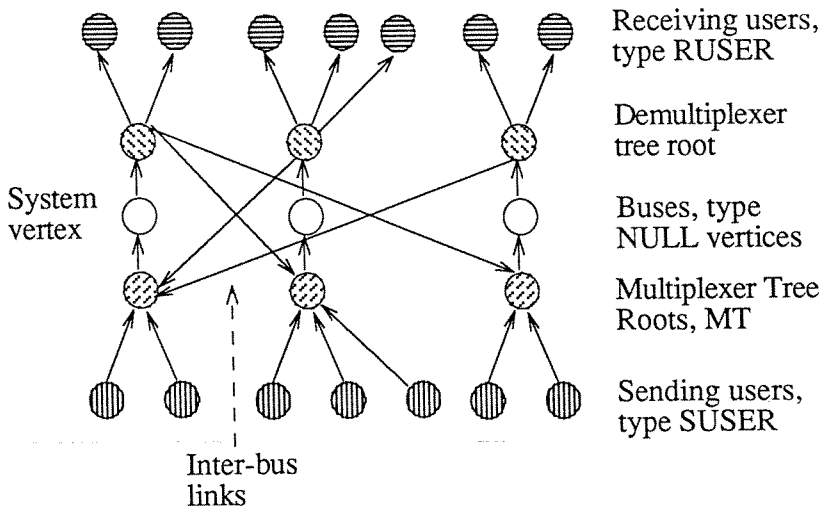

Figure 7.1: AG for the scheduling problem with local and remote transfers, *LocalRemoteDTS*

and Lr is a bijection from Er to T. RG is further restricted in that if the only path between a pair of $SUSER$ and $RUSER$ vertices in AG contains more than one inter-bus link, there is no arc between those vertices in RG.

f is makespan.

Preempt is true.

We discuss two applications which give rise to the *LocalRemoteDTS* problem.

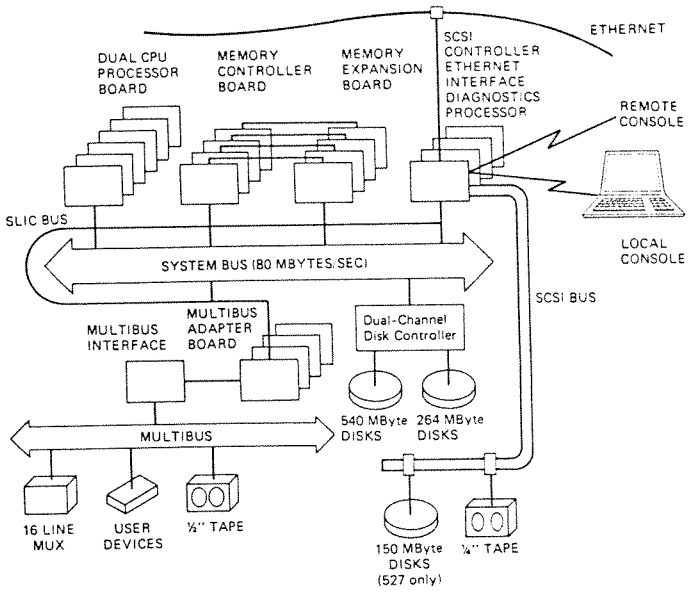

Figure 7.2: Sequent architecture

7.1.2 The parallel I/O application

The bus I/O scheduling problem consists of scheduling the data transfers among processors and peripheral devices connected via a collection of hierarchically organized buses. Examples of such architectures include the commercially succesful Sequent [100] and the research prototype Hector [138]. The Sequent system (Fig. 7.2) has a two-tier arrangement of buses. A single fast system bus connects processors and main memory. Several relatively slow I/O buses, e.g. SCSI or Multibus (see [68]), connect I/O devices to the system bus. Each bus permits at most one data transfer to be in progress at any given time.

Two types of I/O transfers may take place. Data transfers from one I/O device to another on the same bus are called *local* transfers, while those among

main memory and I/O devices on separate buses are called *remote* transfers. A remote transfer requires simultaneous posession of an I/O device, a system bus, an I/O bus, and either a memory unit or another I/O device. A local transfer requires two I/O devices and a local bus. An example of an application requiring local transfers is 3D visualization of scientific data (e.g. [144]), for which a fast disk may supply data to a high-performance graphics workstation connected to a common local bus. A more mundane example is the periodic backup of files from disks to tape. Each transfer consists of a number of fixed-size units (e.g. disk blocks) which we call packets, and transfers can be preempted at packet boundaries. It is assumed that each I/O device or memory unit has one port and is either a sender or a receiver of data.

As mentioned in Chapter 3, given the increasing data transfer demands of new applications programs, there are likely to be multiple parallel buses in future bus-oriented parallel architectures, as in the IBM RP3 [115]. Even if only one bus is available, however, if the bus bandwidth is high enough it can be time-shared to effectively provide many parallel I/O transfers. An example of a bus for which this is possible is the Sequent system bus, which has a bandwidth of 53 MB/s.

7.1.3 The intersatellite communications application

The intersatellite communications scheduling problem [7] consists of scheduling the data transfers among a set of *zones* on the ground via a network of satellites, ground-satellite links and intersatellite links (ISL). Each zone communicates with one satellite. Each ground-satellite link and each ISL is

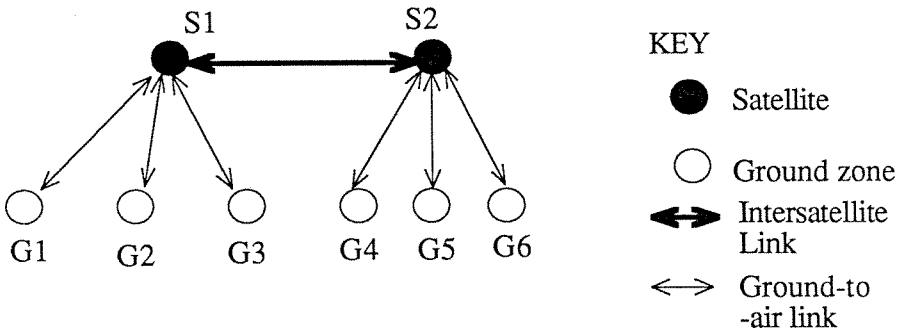

Figure 7.3: Example ISL communications system

bidirectional and allows at most one transfer in each direction at any given time. Both local transfers between zones connected to the same satellite, as well as intersatellite transfers between zones connected to different satellites, are possible (see Fig. 7.3). Data for a single transfer is transmitted as fixed-length packets which may be interspersed on the communications links with packets from other transfers. It is assumed that there is no intersatellite transfer required between two zones if their respective satellites are not connected by an ISL.

Bertossi et al [7] have shown that the ISL communication scheduling problem is NP-complete for an arbitrary number of satellites even for highly restricted ISL network topologies, and zero ISL propagation delay. They conjecture [7] that the special case of just two satellites and zero ISL delay, which we call the *SimpleISL* problem, is also NP-complete, and propose two suboptimal

heuristics. Both heuristics generate schedules of at most twice the optimal schedule length. That is, the upper bound on the makespan, $UB(1) = 2\ LB$, where LB is a lower bound on the makespan defined as follows. Let $L(u,v)$ be the total intersatellite traffic from satellite u to satellite v, $ST(u)$ the total traffic sent from zone u, and $RT(u)$ the total traffic received at zone u. Then,

$$LB = max\{max\{L(i,j) : 1 \leq i,j \leq b \wedge i \neq j\},$$
$$max\{ST(i) : 1 \leq i \leq n\}, max\{RT(i) : 1 \leq i \leq n\}\}$$

The time complexity of the first heuristic is $O(n^{4.5})$, and of the second is $O(n^{8.5})$.

The ISL problem has also been formulated as a modified open-shop by Ganz and Gao [53] for the case of arbitrary ISL propagation delay, and a heuristic has been proposed. The modified open-shop formulation models each uplink as a processor and each downlink as a job. Since each local data transfer requires an uplink and a downlink simultaneously it is modeled as one operation of a job on a processor. Since the transfers through a downlink may occur in any order, the operations of jobs on processors are modeled as not having any technological cobstraints, i.e., as an open shop.

The difficulty with this formulation arises in its modeling of intersatellite transfers, which require three resources (uplink, downlink and ISL) simultaneously. Each direction of an ISL is modeled as a processor and a job consisting of only one operation which executes on that processor. Thus an intersatellite transfer assumes implicitly that two processors are used simultaneously, and

this assumption is enforced when each time slot is scheduled by the heuristic solution [53].

From our viewpoint the problem specification in our model is preferable to the open-shop formulation as the former clearly and explicitly specifies precisely which transfers require which three resources simultaneously. (This approach subscribes to software engineering principles which advocate a clean separation between the specification of a program and its implementation).

An assumption made by Ganz and Gao [53] is that the ISL propagation delay δ is the same for all ISL. In addition, an intersatellite transfer of length t time units is assumed to require the uplink and downlink for $t + \delta$ time units and the ISL for t units. We assume that an intersatellite transfer requires all three links for time $t + \delta$.

SimpleISL is the same as *LocalRemoteDTS* except that $b = 2$, and, since a zone may send and receive data simultaneously, each zone is represented as one $SUSER$ vertex and one $RUSER$ vertex. In the following section we describe a heuristic to solve *LocalRemoteDTS* and *SimpleISL*.

7.1.4 The decomposition heuristic

The heuristic we use to solve *LocalRemoteDTS* is to decompose the architecture graph into trees and apply the solution used for *TreeDTS*. One set of trees allows only local transfers to take place, and one tree allows the remote

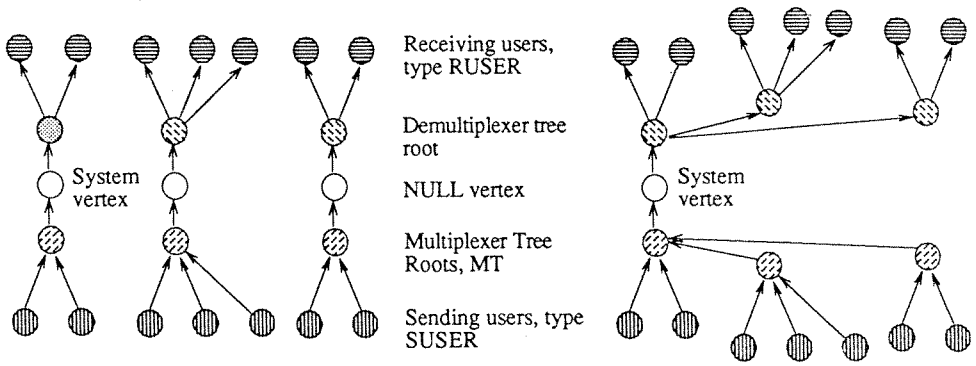

Figure 7.4: Applying the decomposition heuristic

transfers to take place. An example of the decomposition is shown in Fig. 7.4.

Decomposition Heuristic.

Input: Scheduling problem *LocalRemoteDTS.*

Output: A schedule satisfying *LocalRemoteDTS.*

Step A. Decompose LocalRemoteDTS to b Local data transfer problems

1. Let $AG' = (R, Ea', La')$ with $Ea' = Ea - \{e : e \text{ is an inter-bus link}\}$ to to obtain a forest $AG' = \{AG'(1), AG'(2), ..., AG'(b)\}$. La' is La restricted to Ea'.
2. For $1 \leq i \leq b$, let $RG'(i) = (R'(i), Er'(i), Lr'(i))$ be RG restricted to the vertices in $AG(i)$.

3. For $1 \leq i \leq b$, let $PG'(i) = (T'(i), Ep'(i), Lp'(i))$ with $T'(i)$ restricted to the arcs of $Er'(i)$, i.e., $T'(i) = \{t : t \in Lr'(i)\}$, and $Ep'(i)$ and $Lp'(i)$ restricted to $T'(i)$.
4. For $1 \leq i \leq b$, there is a scheduling problem *LocalDTS(i) = (PG(i), AG(i), RG(i), f, Preempt).*

Step B. Decompose LocalRemoteDTS to one Remote data transfer problem

1. Let $AG'' = (R'', Ea'', La'')$ be constructed as follows. Denote the $NULL$ system vertex as s, its MUX child as MT and its $DMUX$ child as DT. Then $R'' = R - \{v : v \in NULL - \{s\}\}$. Let E be Ea restricted to R'' and with all inter-bus links deleted. Let $F = \{(u, MT) : u \neq MT \wedge u$ *is a MUX vertex*$\}$. Similarly let $G = \{(DT, u) : u \neq DT \wedge u$ *is a DMUX vertex*$\}$. Then $Ea'' = E \cup F \cup G$. La'' is La restricted to Ea'' and with unit capacities for arcs in $F \cup G$.
2. Let $RG'' = (R', Er'', Lr'')$ with $Er'' = Er - \{e : e \in Er'(i), 1 \leq i \leq b\}$, and Lr'' restricted to Er''.
3. Let $PG'' = (T'', Ep'', Lp'')$ where $T'' = \{t : t = Lr''\}$, and Ep'' and Lp'' are restricted to T''.
4. Obtain a scheduling problem *RemoteDTS = (PG", AG", RG", f, Preempt).*

Step C. Schedule Local data transfers. For each *LocalDTS(i)*, call the **Tree** algorithm.

Step D. Schedule Remote data transfers. Call the **Tree** algorithm to solve *RemoteDTS.*

End Heuristic.

Theorem 7.1 *The decomposition heuristic solves an instance of LocalRemoteDTS in time $O(n^4)$, providing a schedule with an upper bound on the makespan of $UB(2) = 2\ LB$.*

Proof. The heuristic clearly terminates, and from the construction it is plain that the $b+1$ scheduling problems $LocalDTS(i)$ and $RemoteDTS$ are special cases of *TreeDTS*. Recall that the time complexity of the **Tree** algorithm is $O(n^4C)$ where n is the number of user nodes and C is the average capacity of the user links For *LocalRemoteDTS* the user nodes are $SUSER$ and $RUSER$ vertices and the user links are the arcs incident upon them, so $C = 1$. For $LocalDTS(i)$ the number of user nodes is $2N = n/b$ while for $RemoteDTS$ it is n, so that the time complexity of the decomposition heuristic is $O(n^4)$. The length of the schedule generated by the heuristic has an upper bound given by

$$\begin{aligned} UB(2) &= max\{L(i,j) : 1 \leq i,j \leq b \wedge i \neq j\} \\ &\quad + max\{max\{ST(i) : 1 \leq i \leq n\}, max\{RT(i) : 1 \leq i \leq n\}\} \\ &\leq 2\ LB \\ &= UB(1) \end{aligned}$$

□

The decomposition heuristic is a generalization of the heuristics of Bertossi et al [7]. For the *SimpleISL* problem the decomposition heuristic compares favorably with those Bertossi et al, which have the same upper bound on schedule length and have time complexities of $O(n^{4.5})$ and $O(n^{8.5})$.

Our approach to the scheduling problem for local and remote transfers also illustrates the use of the scheduling model to obtain effective solutions to scheduling problems by graphical decomposition of the abstract specification.

7.2 Systems allowing arbitrary preemption

In this section we briefly note that the **Tree** algorithm can be modified to solve problems in which the traffic demand is non-integer or the system allows preemption at arbitrary boundaries. This may be useful for data transfers involving continuous media, which is becoming more commonly used in multimedia applications.

It turns out that the proof of correctness, and the time complexity, of **Tree** derived in need to be modified only slightly. The lower bound on the schedule length becomes:

$$L'(r) = \max_{e \in E} \frac{t_r(e)}{c(e)}$$

The definition of *network link lower bound* and *network link capacity* are changed to:

$$b_h^*(e) = \begin{cases} c(e), & \text{if } e \in E \text{ and } \frac{t_r(e)}{c(e)} = L'(r) \\ 0, & \text{otherwise} \end{cases}$$

$$c_h^*(e) = \begin{cases} 0, & \text{if } e \in E \text{ and } t_r(e) = 0 \\ c(e), & \text{otherwise} \end{cases}$$

Similar changes to other definitions and proofs, most of which are straightforward, show that **Tree** remains optimal for solving $TreeDTS$ with arbitrary preemptions, and the time complexity remains $O(Cn^4)$. For details, see Sasaki and Jain [125].

7.3 Applications to packet radio and transceiver systems

An important extension to the $TreeDTS$ problem is the case where users communicate through *transceivers*. A transceiver is a device that can transmit and receive, but not both at the same time. This problem has applications for packet radio networks and other communications networks. Special cases of the problem have been studied by Hajek and Sasaki as well as Choi and Hakimi [66, 25].

The paper by Sasaki and Jain [125] shows that the **Tree** algorithm modified to handle arbitrary preemptions, mentioned above, can be used as a subroutine to approximately solve a special case of $TreeDTS$ with transceivers. The modified algorithmm has a time complexity of $O(Cmn^2)$. The proof is due to Sasaki.

7.4 Conclusions and future work

The previous work related to the results in this chapter has already been discussed; it can be found in [7, 66, 25, 53, 54, 125].

Our contributions in this chapter can be summarized as follows. Firstly, we have found an approximation algorithm for the problem of data transfers in tree architectures when both local and remote transfers are allowed. This problem has important applications for parallel I/O as well as intersatellite communications. Our algorithm generalizes previous work that was done in the context of intersatellite communications, provides a performance guarantee at least as good as that provided by the previous best heuristic, and has better time complexity. Secondly, we have shown that the **Tree** algorithm can be modified to apply to tree architectures in which preemption can occur arbitrarily. Such systems may become more prevalent in the future with the spread of continuous media in multimedia systems. Finally, we mention that the **Tree** algorithm has been shown to provide a heuristic for data transfer scheduling in systems with transceivers.

For future work, we suggest two questions. The first is to determine the operating parametres for which the heuristic for local and remote transfers, as well as the **Tree** algorithm for continuous media, provide practical solutions for the parallel I/O application. The second is to resolve the open question of whether *SimpleISL*, the intersatellite communication problem with two satellites for which we have provided a heuristic, is NP-complete.

Chapter 8

Scheduling Tasks Under Mutual Exclusion and Precedence Constraints

A natural extension to the data transfer problem DTS is to allow the specification of logical constraints between tasks. In this chapter we will consider two types of constraints: mutual exclusion constraints and precedence constraints. Informally, a mutual exclusion constraint between two tasks means that in any legal schedule they are not permitted to execute simultaneously. Note that precedence constraints are logically stronger than mutual exclusion constraints, as they additionaly specify the order in which tasks must occur. While the scheduling of tasks with precedence constraints has been studied extensively (e.g. see Chapter 2 and [56, 112]), to our knowledge the previous work on mutual exclusion constraints is very limited, despite its practical applicability to parallel computing. Thus we first concentrate on developing results for scheduling tasks under mutual exclusion constraints, and in sec. 8.2 we present some results on the NP-completeness of precedence-constrained scheduling.

8.1 Mutual exclusion constraints

Mutual exclusion constraints on tasks are very common in parallel programs, and represent a natural and practical means of expressing synchronization requirements. The CODE parallel programming environment, for instance, provides a limited form of mutual exclusion constraints for this purpose [15], [140], [141]. We surmise that mutual exclusion constraints may also be useful for expressing scheduling constraints on problems drawn from many other application areas, including the communications switching application.

An advantage of our model (see Chapter 2) over simpler classification schemes is the ability to perform graph transformations to the problem specifications so as to prove that two problems are related in the sense that a solution to one is a solution to the other. In this section we demonstrate this by showing the scheduling problem for data transfers in systems with tree-structured architectures (*TreeDTS*) is related to scheduling data transfers in the presence of limited mutual exclusion constraints. The limited mutual exclusion constraints are a superset of those allowed in the CODE 1.2 parallel programming environment [140]. We thus obtain an algorithm producing optimal-length schedules for this application of parallel I/O scheduling in a parallel programming environment.

8.1.1 Problem definition

We consider the data transfer scheduling problem in which each data transfer operation involves a distinct pair of resources drawn from two disjoint resource

sets, and each transfer may be required to be logically mutually exclusive with a (restricted) set of other transfers. For simplicity we assume in the following that the architecture provides a direct dedicated link between every pair of communicating entities.

In our model the problem is specified as follows:

$$LimMutex = (PG_1, AG_1, RG_1, f_1, Preempt_1)$$

where the elements of the 5-tuple are defined as follows.

$PG_1 = (T, Ep, Lp)$ where $|T| = n$, $Lp(t)$ is the length of task $t \in T$, and Ep is a set of hyperedges. Recall that in the model hyperedges represent mutual exclusion constraints between tasks, i.e., no two tasks included in the same hyperedge may execute simultaneously. We will introduce a hierarchy of restrictions on Ep below.

$AG_1 = (R, Ea, La)$ with $|R| = 2n$ contains vertices of type $SUSER$ and $RUSER$ corresponding to sending users and receiving users respectively. For ease of exposition we assume the number of both sending users and receiving users is equal to n, although this restriction can easily be relaxed. Ea forms a complete directed bipartite graph from vertices of type $SUSER$ to vertices of type $RUSER$, and $La(e) = 1$ is the link capacity for all $e \in Ea$.

$RG_1 = (R, Er, Lr)$ where Er is a set of arcs from vertices of type $SUSER$ to those of type $RUSER$ such that no two edges share a vertex. Lr is a bijection from Er to T.

f_1 is makespan.

$Preempt_1 = true$.

Terminology. The abbreviation *mutex* stands for "mutual exclusion constraint"; the plural is *mutexes*. Vertices and hyperedges in PG will sometimes be called the entities they represent, i.e., tasks and mutexes, and vice versa. A set of vertices in PG (i.e., tasks) connected by a hyperedge is called a *mutex set*. A task $t \in T$ is said to *participate in a mutex* if it is a member of a mutex set in PG. The degree of a graph G is denoted $degree(G)$.

8.1.2 Limited Mutual Exclusion Constraints

Allowing arbitrary mutexes between tasks can lead to very complex restrictions which are difficult for run-time systems to enforce efficiently, let alone schedule optimally; to our knowledge there are no published scheduling results in this area. It is necessary to consider limited mutexes that provide a trade-off between expressibility and efficiency. We consider three successively looser restrictions.

R1. A task may participate in at most one mutex, i.e., $degree(PG_1) \leq 1$.

R2. There are no hyperedge cycles of odd length in PG_1 and $degree(PG_1) \leq 2$.

R3. A task may participate in at most 2 mutexes, i.e., $degree(PG_1) \leq 2$.

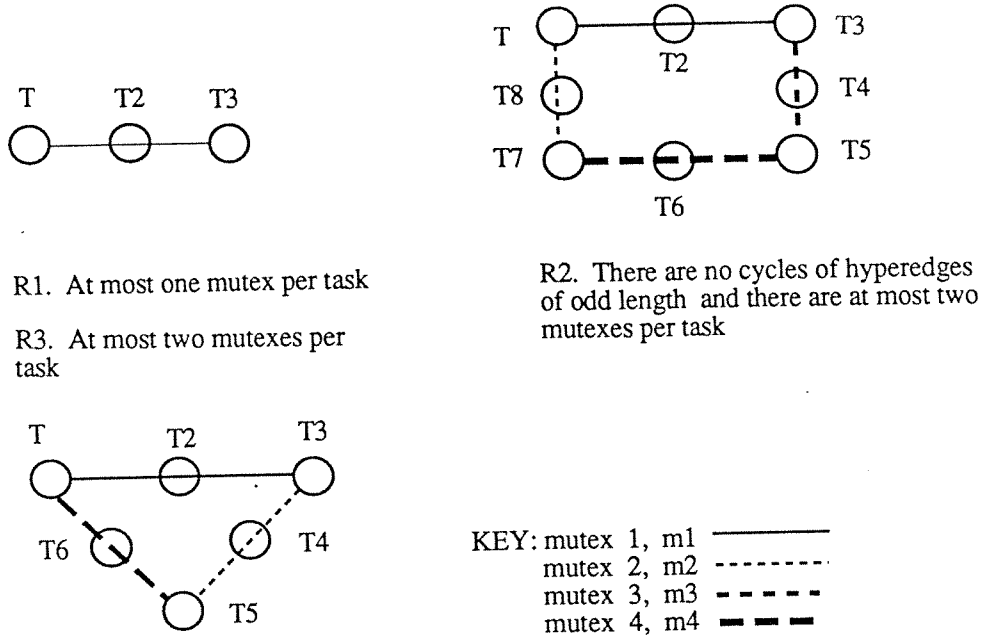

Figure 8.1: Limited mutual exclusion constraints

Clearly, $R1 \Rightarrow R2$, and $R2 \Rightarrow R3$. In Fig. 8.1 we show examples of precedence graphs satisfying *R1 - R3*. Restriction *R1* is implemented in the CODE 1.2 parallel programming environment [140]. We will show that there exist instances of the *LimMutex* problem satisfying *R3*, but not *R2*, which cannot be scheduled using the **Tree** algorithm of Chapter 5.

8.1.3 Transformation

We will transform *LimMutex* into the problem *Tree** defined below, which is an instance of *TreeDTS*. The basic idea is that the mutual exclusion constraints in *LimMutex* are converted to architecture constraints. For an example of this transformation, see Fig. 8.2.

$$Tree^* = (PG_2, AG_2, RG_2, f_2, Preempt_2)$$

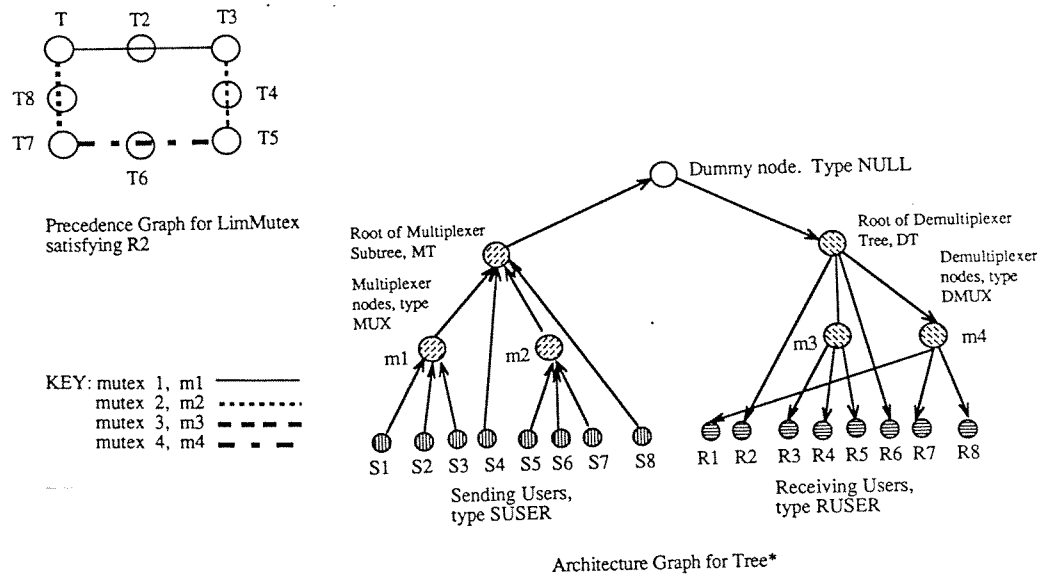

Figure 8.2: Example mutex transformation

where

$PG_2 = (T, Ep, Lp)$ has $|T| = n$ and $Ep = \{\}$, as in *TreeDTS*. $Preempt_2 = Preempt_1$ and $f_2 = f_1$, and both variables are as in *TreeDTS*. $RG_2 = RG_1$, and hence this is a special case of *TreeDTS* where no two arcs share a vertex since there are no resource assignment conflicts. $AG_2 = (R, Ea, La)$ is a special case of *TreeDTS* (see Fig. 5.1) in which all arcs not connected to the root have capacity 1, and the arcs entering and leaving the root have capacity n.

An informal explanation of the transformation is given here (see Theorem 8.1 below for the proof). The basic idea is that the two sets of leaves of the architecture graph of *Tree** represent data senders and receivers respectively, and the interior vertices model the mutual exclusion constraints of *LimMutex*.

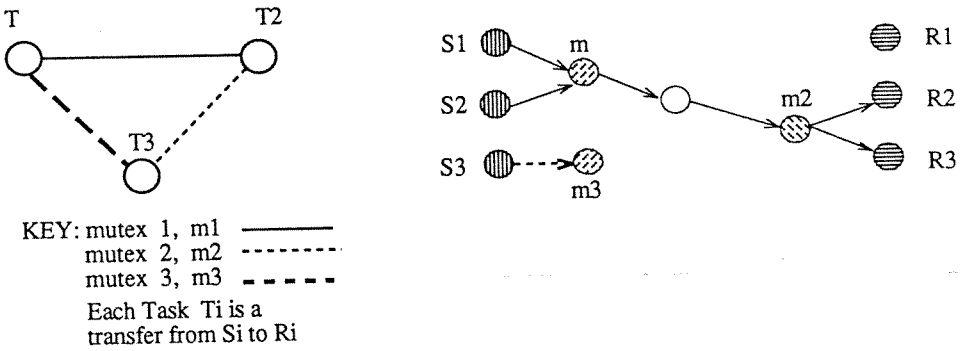

Figure 8.3: Example PG satisfying $R3 \wedge \neg R2$

For instance, a MUX vertex with two incoming unit-capacity arcs from two senders, but a single outgoing arc of unit capacity, will allow only one of the senders to transfer data at any given time.

Both MUX and $DMUX$ interior vertices can be introduced to model mutexes, and each sender and each receiver can be connected to an interior vertex. Since each data transfer task involves a distinct sender-receiver pair, it would thus seem that each task can be allowed to participate in two mutexes; in other words, restriction $R3$ would seem to suffice. However, the tree topology of the architecture graph requires that every path from a sender to a receiver include the root vertex. This necessitates the stronger restriction $R2$ on mutexes. Fig. 8.3 shows an example for which $R3$ is satisfied but $R2$ is violated, and an instance of $Tree^*$ cannot be constructed.

Theorem 8.1 *An instance of LimMutex satisfying R2 can be transformed to an instance of Tree*.*

Proof. (Follows from Lemmas 8.1 - 8.3 below.) For convenience PG_1, the precedence graph of *LimMutex*, is converted to a *mutex graph* containing edges but no hyperedges. Condition *R2* can then be stated as an equivalent condition *R2'* on the mutex graph (see Lemma 8.1). The key step of the proof is contained in Lemma 8.2, which shows that, provided the mutex graph satisfies *R2'*, each vertex in the mutex graph (i.e., each mutex in PG_1) can be systematically represented as an interior vertex of type MUX or $DMUX$ in AG_2 so that adjacent vertices in the mutex graph are assigned different types. This process is called *typing*, and an algorithm for performing it is given. Once the interior vertices have been consistently typed, it is a straightforward construction to obtain AG_2 (see Lemma 8.3). □

Def. Given a precedence graph $G = (V, E, L)$ containing only hyperedges, the corresponding mutex graph is $M_G = (V', E', L')$ where $V' = \{m_i : e_i \in E\}$, $E' = \{(m_i, m_j) : e_i, e_j \in E \wedge e_i \cap e_j \neq \{\}\}$, and $L' : V' \rightarrow 2^V$ is a labeling function such that $L(m_i) = \{u : e_i \textit{ is incident on } u\}$.

Vertices of M_G may also be called mutexes. We state a condition *R2'* on the mutex graph.

R2'. There are no cycles of odd length in the mutex graph MG corresponding to PG_1.

Lemma 8.1 $R2 \equiv R2'$.

Proof. Clearly $degree(PG_1) \geq 3 \Rightarrow \neg R2'$, and if there are cycles of odd length in PG_1 so are there in MG. Hence $\neg R2 \Rightarrow \neg R2$. Proving $\neg R2 \Rightarrow \neg R2$ is equivalent to showing that if MG has an odd cycle and $degree(PG_1) \leq 2$ then PG_1 also has an odd cycle, which follows from the construction of MG. □

We assign a type MUX or $DMUX$ to each vertex in the mutex graph MG using the Typing Algorithm given below. We will see that in order to subsequently construct a tree, the typing algorithm must assign different types to adjacent vertices in the mutex graph, which can only be done if the mutex graph satisfies *R2'*.

Def. A vertex set $M \subseteq V'$ of a mutex graph $MG = (V', E', L')$ is said to be *consistently typed* if for all $m, m' \in M$ such that $(m, m') \in E'$, $type(m) \neq type(m')$. A mutex graph is said to be consistently typed if the set of all its vertices is consistently typed.

Typing Algorithm.

Input. Mutex graph $MG = (V', E', L')$ satisfying *R2'*.

Output. Type function $type : V' \rightarrow \{MUX, DMUX\}$.

Let $type(m_1) = MUX$ for some $m_1 \in V'$

Let $A_1 = (V_1, E_1)$ with $V_1 = \{m_1\}$.

$i = 1$

Repeat

Choose some $m \in V' - V_i$ such that for some $m' \in V_i$, $(m, m') \in E'$.

Let $type(m)$ be the opposite of $type(m')$

Let $A_{i+1} = (V_{i+1}, E_{i+1})$ with$V_{i+1} = V_i \cup \{m\}$ and $E_{i+1} = E_i \cup \{(m, m')\}$.

$i = i + 1$

Until all vertices in V' are typed.

End algorithm.

Lemma 8.2 *Given a mutex graph satisfying R2', the Typing Algorithm produces a consistently typed mutex graph.*

Proof. By induction on the sequence $A_1, ..., A_p$. Wlog assume that the mutex graph $MG = (V', E', L')$ is strongly connected. Then by the construction so are all the A_i.

basis. Clearly A_1 is consistently typed.

hyp. A_i is consistently typed.

ind. Let m be the unique element of $V_{i+1} - V_i$. By construction there exists a vertex $m' \in V_i$ such that $(m, m') \in E'$ and $type(m) \neq type(m')$. Suppose there exists $m'' \in V_i$ such that $(m, m'') \in E'$ and $type(m) = type(m'')$. Then there exists a path of mutexes $(m' = n_1, n_2, ..., n_q = m'')$ in A_i since A_i is

strongly connected; in addition, $type(n_i) \neq type(n_{i+1})$, for $1 \leq i < q$, since A_i is consistently typed. The path $(n_1, n_2, ..., n_q)$ followed by m is a cycle of odd length, contradicting *R2'*. □

Lemma 8.3 *An architecture graph AG_2 satisfying Tree* can be constructed from a consistently typed mutex graph $MG = (V', E', L')$.*

Proof. By construction of $AG_2 = (V_2, E_2, L_2)$.

Vertices of AG_2. Let $V_2 = S \cup R \cup M \cup D \cup \{MT, DT, r\}$ defined as follows. S and R are sets of n vertices each of type $SUSER$ and $RUSER$ respectively. Set $M = \{m_i : m_i \in V' \wedge type(m_i) = MUX\}$, and $D = \{d_i : d_i \in V' \wedge type(d_i) = DMUX\}$. Set the types of the roots of the multiplexer tree, the demultiplexer tree, and of AG_2 as $type(MT) = MUX$, $type(DT) = DMUX$, and $type(r) = NULL$.

Arcs of AG_2. Let $E_2 = E_S \cup E_R \cup E_M \cup E_D \cup E_X \cup \{(MT, r), (r, DT)\}$ defined as follows. Add arcs from sending users to a vertex $m \in M$ if transfers from that user participate in m, i.e., let $E_S = \cup_{m \in M} E_m$ where $E_m = \{(s_i, m) : s_i \in S \wedge v_i \in L'(m)\}$. Similarly let $E_R = \cup_{d \in D} E_d$ where $E_d = \{(d, r_i) : r_i \in R \wedge v_i \in L'(d)\}$. Add arcs from the MUX vertices to the root of the multiplexer subtree, i.e., let $E_M = \{(m, MT) : m \in M\}$. Similiarly, let $E_D = \{(DT, d) : d \in D\}$. Finally, connect sending users whose transfers do not participate in mutexes directly to the root of the multiplexer subtree, and similarly for receivers, i.e., let $E_X = \{(s, MT) : s \in S \wedge degree(s) = 0\} \cup \{(DT, s) : s \in R \wedge degree(s) = 0\}$.

Labels of AG_2. Set the arc capacities of all links except those incident on the $NULL$ vertex to 1, i.e., for all $e \in E_2 - \{(MT, r), (r, DT)\}$ let $L_2(e) = 1$. Let $L_2((MT, r)) = L_2((r, DT)) = n$. □

8.2 Precedence constraints

In this section we consider the problem of scheduling data transfers under the presence of precedence constraints. Clearly, this is an especially important problem for the parallel I/O application. We will show, however, that even for situations in which tasks are of unit length and the precdence constraints are restricted to be in the form of a tree, the problem is NP-complete. In the following we specify the problem and review some well-known related NP-completeness results. We then observe the NP-completeness of our problem, and suggest avenues for further work.

We specify a restricted form of the precedence-constrained data transfer scheduling problem which we will later show to NP-complete.

$TreePrecDTS = (PG, AG, RG, f, Preempt)$

where

$PG = (T, Ep, Lp)$ has, for all $t \in T$, $Lp(t) = 1$, and is a tree.

$AG = (R, Ea, La)$ is a complete bipartite graph.

$RG = (R, Er, Lr)$ is a bipartite graph with $|Er| = |T|$.

f is makespan, $Preempt$ is true.

We contrast the $TreePrecDTS$ problem with two related scheduling problems, one of which is the well-known *Resource Constrained Scheduling* problem for multiprocessors [55, 56], a special case of which we call $TreeRCMS$, specified as follows.

$$TreeRCMS = (PG, AG', RG', f, Preempt)$$

where PG, f and $Preempt$ are as for $TreePrecDTS$.

$AG' = (R, Ea, La)$ is a bipartite graph, i.e., R is partitioned into a set of 'processors', R_p, and a set of 'other resources', R_r, but $Ea = \{\}$.

$RG' = (R, Er, Lr)$ has $R = R_p \cup R_r$ and $Er = \{\}$, i.e., the assignment of tasks to resource instances is not specified. The task resource requirement tr is that each task requires one processor and some number of other resources, i.e., for all $t \in T$, there exists k, $0 < k \leq |R_r|$, such that $tr(t) \in R_p \times R_r^k$.

It is known that $TreeRCMS$ is NP-complete [55]. However, we note that in $TreeRCMS$ the assignment of tasks to resources has to be computed as well as a schedule minimizing the makespan. Therefore it might be possible that if the assignment is fixed, finding the schedule is not NP-complete. In fact,

this turns out not to be the case, as shown by considering a related problem, *Processor-Bound Multiprocessor Scheduling* with tree precedences [64], which we call $TreePBMS$.

$TreePBMS = (PG, AG'', RG'', f, Preempt)$

where PG, f and $Preempt$ are as for $TreePrecDTS$ and $TreeRCMS$.

$AG'' = (R, Ea, La)$ consists of $|R|$ distinguished vertices, with $Ea = \{\}$.

$RG'' = (R, Er, Lr)$ has $|Er| = |T|$ consisting of self-loops on each vertex.

Thus $TreePBMS$ differs from $TreePrecDTS$ and $TreeRCMS$ in that each task requires only one resource, but also differs from $TreeRCMS$ in that the assignment of tasks to resource instances is known. Goyal [64, 56] has shown, using a reduction very similar to that used by Garey and Johnson [55], that $TreePBMS$ is NP-complete.

Observation. $TreePrecDTS$ is NP-complete.

The observation follows from the NP-completeness of $TreePBMS$, and noting that $TreePrecDTS$ is a special case of $TreePBMS$.

8.2.1 Further work

Since the $TreePrecDTS$ problem is of practical interest, it is useful to look for approximation algorithms for its solution. We are currently investigating

several such schemes [79].

A way of specifying the problem that may be useful for future work is to "merge" the resource and precedence graphs. Informally, the precedence graph is augmented by adding a hyperdge connecting any tasks that require the same resource instance (i.e., edges in the resource graph that have a common vertex). Then the edges of the resource graph can be deleted, since resource conflict information is already captured in the extended precedence graph. Note that the extended precedence graph can be simplified: if two vertices are connected by a directed edge, they need not be connected by a hyperedge even if they have a resource conflict. This extended precedence graph may be useful for designing heuristics that consider both the precedence and resource constraints between tasks simultaneously. (It can, of course, also be applied to problems in which there are no resource conflicts but tasks have logical mutual exclusion constraints as well as precedence constraints).

8.3 Discussion

8.3.1 Previous related work

The related work on precedence constrained scheduling [55, 64, 56, 98, and refernces therein] has already been reviewed in sec. 8.2 and Chapter 2.

To our knowledge, there has been no previous work on scheduling of tasks with explicit mutual exclusion constraints. Of course, implicit mutual exclusion constraints, such as situations where every task is mutually exclusive to

every other task since they all require the same resource, have been studied extensively. For instance, some of the recent work on scheduling in the presence of "exclusion constraints" between two tasks (e.g. [145]) actually refers to the restriction that if one task is executing, on a single processor, it may not be preempted by the other. Similarly, general resource constraints implicity define mutual exclusion constraints between tasks (e.g., see [130, 128] and references therein). However, *by specifying the mutual exclusion constraints implicitly, their logical structure is not apparent and cannot be exploited.* Thus general resource-constrained scheduling is NP-complete for the non-preemptive case and requires high-degree polynomial linear programming solutions in the preemptive case [56, 130]. In contrast, by considering the structure of explicit mutual exclusion constraints, we are able to specify a hierarchy of constraints that can occur in practice, and obtain a polynomial-time solution.

Almost all the previous work on logical constraints between tasks has focused on precedence constraints, which are logically stronger than mutual exclusion constraints and do not capture the synchronization requirements of tasks in some applications, particularly parallel programming. Some recent work has started to address this issue by weakening precedence constraints. Berger and Cowen [6] consider tasks which may be subject to precedence constraints (the usual partial order), as well as "concurrency constraints" (tasks that must be scheduled in the same time step) and "weak precedence constraints" (tasks that must be scheduled before, or at the same step as, some other task). They do not consider mutual exclusion constraints and simultaneous resource requirements, however.

8.3.2 Conclusions and further work

We have presented NP-completeness results for the problem of scheduling data transfers under the presence of tree-sructured precedence constraints.

We have also presented a solution to the problem of scheduling data transfer tasks when the tasks are subject to a restricted set of logical mutual exclusion constraints. Such constraints arise naturally in the parallel I/O application, and to our knowledge have not been previously systematically studied. While our results are limited to mutual exclusion constraints of a restricted class, they do apply to those allowed in the CODE 1.2 parallel programming environment. Further, the technique used in this chapter, of systematically transforming formal problem specifications, is promising and likely to be applicable to more general classes of mutual exclusion constraints, as well as other problems.

For further work, we suggest two questions. The first is whether the optimal algorithm we have developed, or a fast approximation algorithm based upon it, could be used in parallel programming environments such as CODE. The situation is quite promising since the mutual exclusion constraints are explicitly and deterministically specified by the user, and since an automatic programming system generates all the synchronization code, the algorithm could be used without the user having to be aware of it. The second question is whether tasks with mutual exclusion constraints could be combined with tasks under the constraints of Berger and Cowen's model [6], i.e., "concurrency constraints" and "weak precedence constraints". This would further

weaken the model of task interaction (compared to the usual model of partial orders, i.e., precedence constrained tasks) and allow scheduling in more realistic situations for applications such as parallel I/O.

Chapter 9

Conclusions and Further Work

We have studied the scheduling of data transfers, a problem of increasing importance in high-performance parallel computers and communications systems, particularly with the advent of advanced applications such as volume visualization and multimedia information systems.

Data transfer scheduling gives rise to an important and interesting class of simultaneous resource scheduling problems. We have defined a general graph-theoretical model for precisely specifying and classifying scheduling problems, and demonstrated its coverage of a wide range of traditional and simultaneous multiple resource scheduling problems. We have used the model for the recognition of the similarity of seemingly different problems from different application areas, for the systematic transformation of one problem specification into that of a seemingly different problem, and for the systematic decomposition of a problem specification into solvable subproblems.

We have obtained optimal and approximate algorithms for a wide range of problems, including communication architectures in which resources are fully

connected, communication architectures with a tree topology, and tree architectures in which both local and remote data transfers are permitted. We have also obtained results for scheduling data transfers under the presence of mutual exclusion constraints and precedence constraints. All these results either solve more general instances of the scheduling problem, or have better time complexity, or provide better approximations than previously known solutions, or all three. Finally, we have undertaken extensive experimenatal evaluations of our algorithms and determined the situations under which they operate best.

The results we have obtained are generally applicable to both parallel computers and communications systems. Specifically, they are applicable to certain types of shared-bus multiprocessor systems such as the Sequent [100], Encore [143], and the IBM RP3 [115]; bus-oriented local area networks such as the Ethernet; TDMA satellite switches [75]; hierarchical switching systems [39]; tree-structured multiprocessor architectures such as the Sequent [100], Tree Machine [133], KYKLOS [102], and Hector [138]; and intersatellite communications systems [7].

For future work, we have posed specific questions at the end of each chapter that relate to the topics studied in that chapter. Here we state some questions of broader practical and theoretical concern.

1. What is the range of architectures and exclusion constraints for which we can obtain optimal, polynomial-time solutions? In particular, can architectures such as the hypercube and mesh-based systems be covered? Is it possible to design and evaluate faster near-optimal solutions?

2. Is it possible to integrate data partitioning and allocation with data transfer scheduling so as to provide a better comprehensive approach to managing parallel I/O?
3. Is it desirable to integrate routing and scheduling of data transfers, in computer networks and multiprocessor architectures such as the hypercube? Is it possible to exploit the similarity of the solution techniques used for some routing and scheduling problems?
4. Can effective parallel algorithms be developed to perform data transfer scheduling?
5. Can the reasoning about the equivalence and transformation of scheduling problem classes using our scheduling model be formalized further into inference rules or general theorems?

BIBLIOGRAPHY

[1] J. Akella and D. P. Siewiorek. Modeling and measurement of the impact of Input/Output on system performance. In *Proc. 18th Intl. Symp. Comp. Arch.*, pages 390–399, 1991.

[2] M. Arrott and S. Latta. Perspectives on visualization. *IEEE Spectrum*, pages 61–65, Sep. 1992.

[3] Kenneth R. Baker. *Introduction to sequencing and scheduling.* John Wiley, 1974.

[4] H. Balan. Master's thesis, Dept. of Elect. and Comp. Eng., Univ. of Texas at Austin, 1990.

[5] Claude Berge. *Graphs.* North-Holland, 1985.

[6] B. Berger and L. Cowen. Complexity results and algorithms for $\{<, \leq, =\}$-constrained scheduling. In *Proc. Symp. on Discrete Alg.*, pages 137–147, 1991.

[7] A. A. Bertossi, G. Bongiovanni, and M. A. Bonuccelli. Time slot assignment in SS/TDMA systems with intersatellite links. *IEEE Trans. Comm.*, 35:602–608, June 1987.

[8] J. A. Bondy and U. S. R. Murty. *Graph theory with applications.* North-Holland, 1976.

[9] G. Bongiovanni, D. Coppersmith, and C. K. Wong. An optimum time slot assignment algorithm for an SS/TDMA system with variable number of transponders. Technical Report RC 8301 (# 35888), IBM T. J. Watson Research Center, 1980.

[10] G. Bongiovanni, D. Coppersmith, and C. K. Wong. An optimum time slot assignment algorithm for an SS/TDMA system with variable number of transponders. *IEEE Trans. Comm.*, 29(5):721–726, May 1981.

[11] M. A. Bonuccelli. A fast time slot assignment algorithm for TDM hierarchical switching systems. *IEEE Trans. Comm.*, 37:870–874, Aug. 1989.

[12] H. Boral and D. J. DeWitt. Database machines: An idea whose time has passed? A critique of the future of database machines. In *Third Intl. Workshop on Database Machines*, pages 166–187, 1983.

[13] H. Boral and P. Faudemay, editors. *Database machines.* Springer-Verlag, 1989.

[14] D. Bradley and D. A. Reed. Performance of the Intel iPSC/2 input/output system. In *Proc. Conf. on Hypercubes, Concurrent Comp. and Appl.*, pages 141–144, 1990.

[15] J. C. Browne, Muhammad Azam, and Stephen Sobek. CODE: A unified approach to parallel programming. *IEEE Software*, page 11, July 1989.

[16] J. C. Browne, A. Dale, C. Leung, and R. Jenevein. A parallel multistage I/O architecture with self-managing disk cache for database management applications. In *Fourth Intl. Workshop on Database Machines.* Springer-Verlag, 1985.

[17] J. C. Browne, G. E. Onstott, P. L. Soffa, Ron Goering, S. Sivaramakrishnan, Harish Balan, and Kiran Somalwar. Design and evaluation of external memory architectures for multiprocessor computer systems: Second quarter report to IBM Yorktown Heights Research Lab. Technical report, Univ. Texas at Austin, Dept. of Comp. Sci., 1987. Available from J. C. Browne.

[18] C. E. Catlett. Balancing resources. *IEEE Spectrum*, pages 48–55, Sep. 1992.

[19] S. Chalasani and A. Varma. Fast parallel time-slot assignment algorithms for TDM switching. In *Proc. Intl. Conf. Par. Proc.*, volume III, page 154, 1990.

[20] S. Chalasani and A. Varma. An improved time slot assignment algorithm for TDMA hierarchical switching systems. In *Proc. Fourth Intl. Conf. Data Comm. Sys. and their Perf.*, pages 116–132, 1990.

[21] W.-T. Chen and H.-J. Liu. An adaptive scheduling algorithm for TDM switching systems. In *Proc. IEEE Infocom*, pages 668–677, 1991.

[22] W.-T. Chen, P.-R. Sheu, and J.-H. Yu. Time slot assignment in TDM multicast switching systems. In *Proc. IEEE Infocom*, 1991.

[23] H.-A. Choi and S. L. Hakimi. Scheduling file transfers for trees and odd cycles. *SIAM J. Comput.*, 16(1):162–168, 1987.

[24] H.-A. Choi and S. L. Hakimi. Data transfers in networks. *Algorithmica*, 3:223–245, 1988.

[25] H.-A. Choi and S. L. Hakimi. Data transfers in networks with transcievers. *Networks*, 18:223–251, 1988.

[26] E. F. Codd. Multiprogram scheduling. *Comm. ACM*, 3, June 1960.

[27] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and A. S. LaPaugh. Scheduling file transfers. *SIAM J. Comput.*, 3:744–780, 1985.

[28] E. G. Coffman, Jr. and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Inf.*, 1:200–213, 1972.

[29] R. Cole and J. Hopcroft. On edge coloring bipartite graphs. *SIAM J. Comput.*, 11(3):540–546, 1982.

[30] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling.* Addison-Wesley, 1967.

[31] D. de Werra. An introduction to timetabling. *European J. of Operational Res.* 19:151–162, 1985.

[32] T. A. DeFanti, M. D. Brown, and B. H. McCormick. Visualization: Expanding scientific and engineering research opportunities. *IEEE Computer*, pages 12–26, Aug. 1989.

[33] P. J. Denning. Effects of scheduling on file memory operations. In *Proc. AFIPS Spring Joint Comp. Conf.*, pages 9–21, 1967.

[34] N. Deo. *Graph Theory with Applications to Engineering and Computer Science.* Prentice-Hall, 1974.

[35] D. J. DeWitt. DIRECT - A multiprocessor organization for supporting relational database management systems. *IEEE Trans. Comp.*, June 1979.

[36] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. Kumar, and M. Muralikrishna. GAMMA - A high performance dataflow database architecture. In *Proc. 12th Intl. Conf. on Very Large Data Bases*, Aug. 1986.

[37] G. Dobson and U. Karmarkar. Simultaneous resource scheduling to minimize weighted flow times. *Oper. Res.*, 37(4):592–600, 1989.

[38] E. W. Dusio, T. P. Murphy, and W. F. Cashman. Communications satellite software: A tutorial. *IEEE Computer*, pages 21–34, Apr. 1991.

[39] K. Y. Eng and A. S. Acampora. Fundamental conditions governing TDM switching assignments in terrestial and satellite networks. *IEEE Trans. Comm.*, COM-35:755–761, 1987.

[40] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM J. Comput.*, 5(4):691–703, 1976.

[41] S. Fiorini and R. J. Wilson. *Edge-colourings of graphs.* Pitman, London, U.K., 1977.

[42] L. R. Ford and D. R. Fulkerson. *Flows in networks.* Princeton University Press, 1962.

[43] E. A. Fox, editor. *CACM Special Issue on Digital multimedia systems.* ACM, Apr. 1991.

[44] J. C. French, T. W. Pratt, and M. Das. Performance meaurement of a parallel Input/Output system for the Intel iPSC/2 hypercube. In *Proc. SIGMETRICS*, pages 178–187, 1991.

[45] Simon French. *Sequencing and Scheduling.* John Wiley, 1982.

[46] A. M. Frieze. Probabilistic analysis of graph algorithms. In G. Tinhofer, E. Mayr, H. Noltemeir, and M. Syslo, editors, *Computational graph theory*, pages 209–233. Springer-Verlag, 1990. Also as *Computing Supp.*, vol. 7, Springer-Verlag, 1990.

[47] M. Fujii, T. Kasami, and K. Ninomiya. Optimal sequencing of two equivalent processors. *SIAM J. App. Math*, 17:784–789, 1969. Erratum, SIAM J. App. Math., vol. 20, p. 141, 1971.

[48] H. Gabow. Using euler partitions to edge color bipartite multigraphs. *Intl. J. Computer and Inf. Sci.*, 5:345–355, 1976.

[49] H. Gabow. An almost linear algorithm for two-processor scheduling. *J. Ass. Comp. Mach.*, 29:766–780, 1982.

[50] H. Gabow and O. Kariv. Algorithms for edge coloring bipartite multigraphs. *ACM Symp. Th. of Comp.*, pages 184–192, 1978.

[51] H. Gabow and O. Kariv. Algorithms for edge coloring bipartite graphs and multigraphs. *SIAM J. Comput.*, 11(1):117–129, 1982.

[52] H. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proc. 15th Ann. Symp. Theory of Comp.*, pages 246–251, 1983.

[53] A. Ganz and Y. Gao. Scheduling on SS/TDMA systems with intersatellite links. In *Proc. Intl. Conf. Comm.*, volume 1, pages 515 - 519, 1989.

[54] A. Ganz and Y. Gao. TDMA communication for SS/TDMA satellites with optical intersatellite links. In *Proc. Intl. Conf. Comm.*, pages 1081 - 1085, 1990.

[55] M. Garey and D. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM J. Comput.*, 4:397, Dec. 1975.

[56] M. Garey and D. Johnson. *Computers and intractability: A guide to the theory of NP-completeness.* Freeman, 1979.

[57] J. Ghosh and B. Agarwal. Parallel I/O subsystems for hypercube multicomputers. In *Proc. Intl. Par. Proc. Symp.*, pages 381–384, 1991.

[58] Alan Gibbons. *Algorithmic graph theory.* Cambridge University Press, 1985.

[59] G. A. Gibson. *Redundant disk arrays: Reliable, parallel secondary storage.* PhD thesis, Univ. of Calif., Berkeley, Comp. Sci. Div, 1990. Also available as Tech. Rep. UCB/CSD 91/613.

[60] Mario Gonzalez, Jr. Deterministic processor scheduling. *Computing Surveys*, 9:173, Sept. 1977.

[61] T. Gonzalez and D. B. Johnson. A new algorithm for preemptive scheduling of trees. *J. Ass. Comp. Mach.*, 27:287–312, 1980.

[62] T. Gonzalez and S. Sahni. Open shop scheduling to minimize finish time. *J. Ass. Comp. Mach.*, 23:665–679, 1976.

[63] C. C. Gotlieb. The construction of class-teacher timetables. In *Proc. IFIP Congress*, pages 73–77, 1962.

[64] D. K. Goyal. Scheduling processor bound systems. Technical Report CS-76-036, Washington State Univ., 1976.

[65] H. Hadimioglu and R. J. Flynn. The architectural design of a tightly-coupled distributed hypercube file system. In *Proc. Conf. on Hypercubes, Concurrent Comp. and Appl.*, pages 147–150, 1989.

[66] B. Hajek and G. Sasaki. Link scheduling in polynomial time. *IEEE Trans. Info. Th.*, 34:910–917, 1988.

[67] B. Hancock. Multiprocessors are NOT always better. *Digitial Rev.*, page 59, Dec. 2 1991.

[68] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[69] D. S. Hochbaum, T. Nishizeki, and D. B. Shmoys. A better than "best possible" algorithm to edge color multigraphs. *SIAM J. Comput.*, 7:79–104, 1986.

[70] I. J. Holyer. The NP-completeness of edge colourings. *SIAM J. Comput.*, 10:718–720, 1980.

[71] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2:225–231. 1973.

[72] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848, 1961.

[73] IEEE. *Proc. Intl. Conf. Univ. Pers. Comm.*, 1992. Held Sep. 29 - Oct. 2, 1992, at Dallas, TX.

[74] T. Inukai. An efficient SS/TDMA time slot assignment algorithm. *IEEE Trans. Comm.*, COM-27:1449–1455, 1979.

[75] Y. Ito, Y. Urano, T. Muratani, and M. Yamaguci. Analysis of a switch matrix for an SS/TDMA system. *Proc. IEEE*, 65:411–419, 1977.

[76] Ravi Jain. Scheduling I/O in parallel computing environments. Unpublished manuscript, Dec. 1990.

[77] Ravi Jain and Galen Sasaki. Scheduling packet transfers in a class of TDM hierarchical switching systems. In *Proc. Intl. Conf. Comm.*, 1991.

[78] Ravi Jain, Kiran Somalwar, John Werth, and J. C. Browne. Scheduling parallel I/O operations in multiple-bus systems. *J. Par. and Distrib. Comp.*, Dec. 1992. Special Issue on Scheduling and Load Balancing.

[79] Ravi Jain and John Werth. Precedence constrained I/O scheduling. Unpublished manuscripts, 1992.

[80] Ravi Jain, John Werth, and J. C. Browne. A general model for scheduling of parallel computations and its application to parallel I/O operations. In *Proc. Intl. Conf. Par. Proc.*, 1991.

[81] Ravi Jain, John Werth, J. C. Browne, and G. Sasaki. A graph-theoretic model for the scheduling problem and its application to simultaneous resource scheduling. In *ORSA Conf. on Computer Science and Operations Research: New Developments in their Interfaces*, Jan. 1992. Available from Pergamon Press.

[82] W. Jilke. Disk array mass storage systems: The new opportunity. Technical report, Amperif Corp., Sep. 1986.

[83] C. V. Jones. The three-dimensional Gantt chart. *Oper. Res.*, 36(6):891–903, 1988.

[84] H. Jordan. Scalability of data transport. In *Proc. Scalable High Perf. Computing Conf.*, pages 1–8, 1992.

[85] A. Kandappan. Data allocation and scheduling for parallel I/O systems. Master's thesis, Dept. of Elect. and Comp. Eng., Univ. of Texas at Austin, 1990.

[86] H. J. Karloff and D. B. Shmoys. Efficient parallel algorithms for edge coloring problems. *J. Algorithms*, 8:39–52, 1987.

[87] K. N. Karna and E. W. Dusio. Communications satellite software. *IEEE Computer*, pages 15–16, Apr. 1983. Special Issue on communications satellite software.

[88] M. Y. Kim. Synchronized disk interleaving. *IEEE Trans. Comp.*, C-35, 1986.

[89] S. J. Kim and J. C. Browne. A general approach to mapping of parallel computations upon multiprocessor architectures. In *Proc. Intl. Conf. Par. Proc.*, pages 1–8, 1988.

[90] T. Kwok. Communications requirements of multimedia applications: A preliminary study. In *Proc. Intl. Conf. Univ. Pers. Comm.*, 1992.

[91] S. Lam and R. Sethi. Worst case analysis of two scheduling algorithms. *SIAM J. Comput.*, 6:518–536, 1977.

[92] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[93] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Recent developments in deterministic sequencing and scheduling: A survey. In

Deterministic and Stochastic Scheduling, pages 35–73. D. Reidel Publishing, 1982.

[94] P. L'Eculyer. Efficient and portable combined random number generators. *Comm. ACM*, 31:742–774, June 1988.

[95] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Oper. Res.*, pages 22–35, 1978.

[96] S. C. Liew. Comments on "Fundamental conditions governing TDM switching assignments in terrestrial and satellite networks". *IEEE Trans. Comm.*, 37:187–189, Feb. 1989.

[97] M. Livny, S. Khoshhafian, and H. Boral. Multi-disk management algorithms. In *Proc. SIGMETRICS*, May 1987.

[98] E. L. Lloyd. Concurrent task systems. *Oper. Res.*, 29:189–201, 1981.

[99] C. Lo, R. S. Wolff, and R. C. Bernhardt. An estimate of network database transaction volume to support universal personal communications services. In *Proc. Intl. Conf. Univ. Pers. Comm.*, 1992.

[100] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proc. Intl. Conf. Par. Proc.*, pages 303–310, 1988.

[101] Weizhen Mao. Directed file transfer scheduling. Submitted for publication, 1992.

[102] B. Menezes and R. Jenevein. KYKLOS: A linear growth fault-tolerant interconnection network. In *Proc. Intl. Conf. Par. Proc.*, pages 498–502, 1985.

[103] E. Miller. Input/Output behavior of supercomputing applications. Technical Report UCB/CSD 91/616, Univ. California, Berekeley, 1991.

[104] W. D. Moren. Disk array: You know it when you see it. *Workstation News*, Apr. 1992.

[105] B. M. E. Moret, 1992. Private communication.

[106] B. M. E. Moret and H. D. Shapiro. *Algorithms from P to NP, Volume 1: Design and efficiency.* Benjamin-Cummings, 1991.

[107] T. N. Mudge, J. P. Hayes, and D. C. Winsor. Multiple bus architectures. *Computer*, 20(6):42–48, June 1987.

[108] R. R. Muntz and E. G. Coffman, Jr. Optimal preemptive scheduling on two-processor systems. *IEEE Trans. Comp.*, C-18:101, 1969.

[109] R. R. Muntz and E. G. Coffman, Jr. Preemptive scheduling of time tasks on multiprocessor systems. *J. Ass. Comp. Mach.*, 17:324–338, 1970.

[110] G. M. Nielson, editor. *IEEE Computer Special Issue on Scientific Visualization.* IEEE, Aug. 1989.

[111] R. G. Ogier. A decomposition method for optimal link scheduling. In *Proc. Allerton Conf. Comput. Comm*, pages 822–823, 1986.

[112] Krishna Palem. *On the complexity of precedence constrained scheduling.* PhD thesis, Univ. Texas at Austin, Dept. of Comp. Sci., 1986. Available as Tech. Rept. TR-86-11.

[113] D. A. Pattreson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. SIGMOD*, 1988.

[114] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Trans. Comp.-Aided Design*, pages 661–679, 1989.

[115] G. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor (RP3): Introduction and architecture. In *Proc. Intl. Conf. Par. Proc.*, pages 764–771, 1985.

[116] P. Pierce. A concurrent file system for a highly parallel mass storage system. In *Proc. Conf. on Hypercubes, Concurrent Comp. and Appl.*, pages 155–160, 1989.

[117] A. Pizzarello and F. Golshani. In-memory databases: An industry perspective. In *Proc. Workshop on Res. Iss. in Data Eng.*, pages 96–101, 1992.

[118] T. Pratt, J. French, P. Dickens, and Jr. S. Janet. A comparison of the architecture and performance of two parallel file systems. In *Proc. Conf. on Hypercubes, Concurrent Comp. and Appl.*, pages 161–166, 1989.

[119] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical recipes: The art of scientific computing.* Cambridge, 1986.

[120] W. Rash. Multimedia moves beyond the hype. *Byte*, pages 85–87, Feb. 1992.

[121] A. L. N. Reddy and P. Banerjee. Design, analysis and simulation of I/O architrctures for hypercube multiprocessors. *IEEE Trans. Par. and Distrib. Sys.*, pages 140–151, Apr. 1990.

[122] R. T. Rockafellar. *Network flows and monotropic optimization.* John Wiley, 1984.

[123] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proc. Euro. Conf. on Art. Intel. (ECAI90)*, 1990.

[124] K. Salem and H. Garcia-Molina. Disk striping. In *Proc. IEEE Intl. Conf. Data Eng.*, 1986.

[125] Galen Sasaki and Ravi Jain. Scheduling data transfers in preemptive hierarchical switching systems, 1991. Submitted to IEEE Trans. Comm.

[126] Galen Sasaki and Ravi Jain. Scheduling data transfers in preemptive hierarchical switching systems with applications to packet radio networks. In *Proc. Infocom*, 1991.

[127] R. K. Schultz and R. J. Zingg. Response time analysis of multiprocessor computers for database support. *ACM Trans. Database Sys.*, pages 14–17, 1984.

[128] Chia Shen, K. Ramamritham, and J. A. Stankovic. Resource reclaiming in real time. *IEEE Real-Time Sys. Symp.*, pages 41–50, 1990.

[129] A. Silberschatz and J. Peterson. *Operating systems concepts.* Addison-Wesley, 1988.

[130] R. Slowinski and J. Weglarz. *Advances in project scheduling.* Elsevier Science Pub., Amsterdam, 1989.

[131] J. E. Smith, W. C. Hsu, and C.Hsuing. Future general purpose supercomputer architectures. In *Proc. Supercomp. '90*, pages 796–804, 1990.

[132] Kiran Somalwar. Data transfer scheduling. Technical Report TR-88-31, Univ. Texas at Austin, Dept. of Comp. Sci., 1988.

[133] S. W. Song. A highly concurrent tree machine for database applications. In *Proc. Intl. Conf. Par. Proc.*, pages 259–268, 1980.

[134] J. D. Ullman. NP-complete scheduling problems. *J. Computer and Sys. Sci.*, 10:384–393, 1975.

[135] J. D. Ullman. Complexity of scheduling problems. In E. G. Coffman, Jr., editor, *Computer and job-shop scheduling theory.* John Wiley, 1976.

[136] A. Varma and S. Chalasani. An incremental time-slot assignment algorithm for TDM hierararchical switching systems. In *Proc. IEEE Intl. Conf. Comm.*, pages 1554–1558, 1991.

[137] V. G. Vizing. On an estimate of the chromatic class of a p-graph. *Diskret. Analiz.*, 3:25–30, 1964. In Russian. See Gabow, 1976.

[138] Z. G. Vranesic, M. Stumm, D. M. Lewis, and R. White. Hector: A hierarchically structured shared-memory multiprocessor. *Computer*, pages 72–79, Jan. 1991.

[139] S. B. Weinstein. *IEEE Spectrum*, 1985.

[140] John Werth, Dwip Banerjee, J. C. Browne, Ravi Jain, Steve Lin, Peter Newton, Ravi Rao, and Steve Sobek. CODE 1.2 User Manual and Tutorials. Technical Report TR-90-35, Univ. Texas at Austin, Dept. of Comp. Sci., November 1990.

[141] John Werth, J. C. Browne, Steve Sobek, T. J. Lee, Peter Newton, and Ravi Jain. The interaction of the formal and practical in parallel

programming environment development: CODE. Technical Report TR-91-09, Univ. Texas at Austin, Dept. of Comp. Sci., 1991.

[142] Jennifer Whitehead. The complexity of file transfer scheduling with forwarding. *SIAM J. Comput.*, 19(2):222–245, Apr. 1990.

[143] A. W. Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *14th Intl. Symp. Comp. Arch.*, pages 244–252, 1987.

[144] R. H. Wolfe, Jr. and C. N. Liu. Interactive visualization of 3D seismic data: A volumetric method. *IEEE Comp. Graphics Appl.*, pages 24–30, July 1988.

[145] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. Soft. Eng.*, 16:360–369. Mar. 1990.

[146] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Trans. Comp.*, page 949, Aug. 1987.

VITA

Ravi Jain was born on July 3, 1960, at Simla, India. After completing high school in Kitwe, Zambia, he received the B.Sc. in Electronics Engineering from The City University, London, in 1980. He obtained an M.S.E.E. from Penn State University in 1982, where his research focused on modeling energy deposition in the auroral ionosphere. From 1982 to 1985 he worked at Syntrex Inc. on communications and systems software for a microcomputer system, a local area network, and a fault-tolerant file server. He later worked at SES Inc. on performance modeling of communications systems, and at the Schlumberger Laboratory for Computer Science on high-level parallel programming.

Jain has been an MCD Fellow at the University of Texas at Austin. His research interests include resource management in parallel and distributed computers, communications protocols, discrete algorithms, and performance analysis. Jain has several refereed publications, and has served as a referee for numerous conferences and journals. Jain is a member of the Upsilon Pi Epsilon and Phi Kappa Phi honorary societies, as well as ACM, IEEE, and CPSR. Jain's current address is Bellcore, 445 South Street, Morristown, NJ 07962.

Permanent address: 114 East 31st St., #311
Austin, TX 78705