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

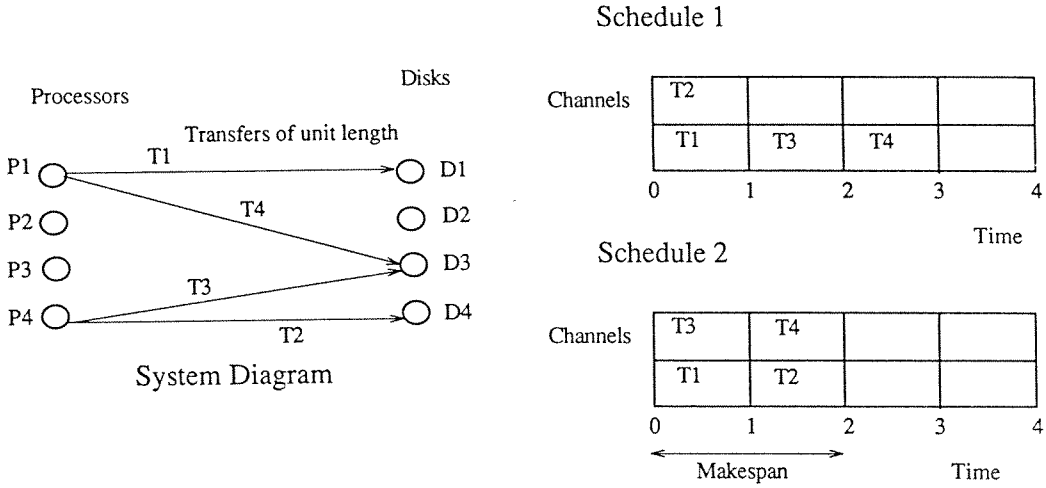

Figure 1.1: Parallel I/O scheduling example

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

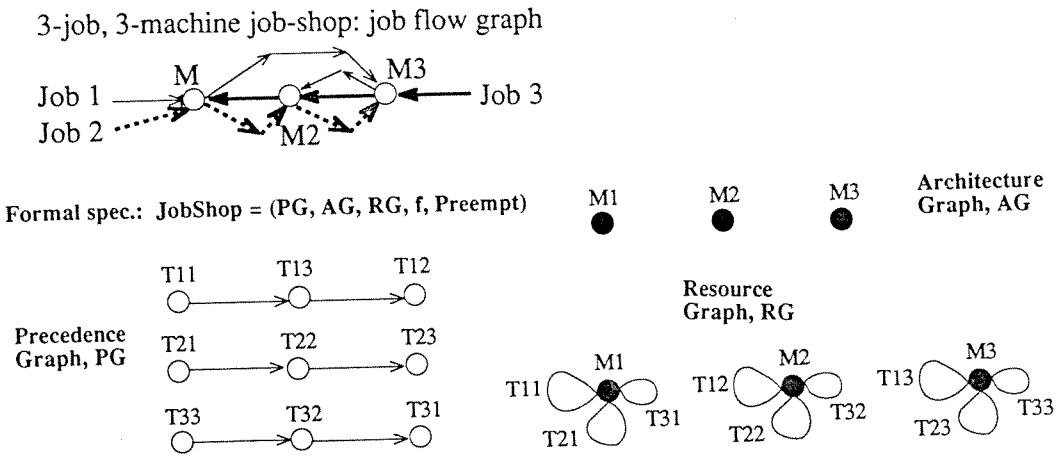

Figure 2.1: Specification of a job-shop example

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.