1986

Multiprocessor scheduling with practical constraints

Kenneth Burton Donovan
University of Central Florida

MULTIPROCESSOR SCHEDULING WITH PRACTICAL CONSTRAINTS

by

KENNETH BURTON DONOVAN

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
the Department of Computer Science
the University of Central Florida
Orlando, Florida

May 1986

Major Professor: Dr. Amar Mukherjee

ABSTRACT

The problem of scheduling tasks onto multiprocessor systems has increasing practical importance as more applications are being addressed with multiprocessor systems. Actual applications and multiprocessor systems have many characteristics which become constraints to the general scheduling problem of minimizing the schedule length. These practical constraints include precedence relations and communication delays between tasks, yet few researchers have considered both these constraints when developing schedulers.

This work examines a more general multiprocessor scheduling problem, which includes these practical scheduling constraints, and develops a new scheduling heuristic using a list scheduler with dynamically computed priorities. The dynamic priority heuristic is compared against an optimal scheduler and against other researchers' approaches for thousands of randomly generated scheduling problems. The dynamic priority heuristic produces schedules with lengths which are 10% to 20% over optimal on the average. The dynamic priority heuristic performs better than other researchers' approaches for scheduling problems with the practical constraints. We conclude that it is important to consider practical constraints in the design of a scheduler and that a simple heuristic can still achieve good performance in this area.

ACKNOWLEDGMENTS

I would like those who helped me - my family, my teachers, my work colleagues and my friends. I am very grateful to General Electric, especially all my managers who were flexible with finances and work schedules to support my graduate studies. Dr. Mukherjee and my graduate committee have provided continual assistance as I developed an initial concept into a complete dissertation. And with my wife Martha, who has offered support and encouragement in all my efforts, I share this accomplishment which came from us both.

TABLE OF CONTENTS

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER 1 PRACTICAL MULTIPROCESSOR SCHEDULING 1
1.1 Scope 1
1.2 Problem Area and Example 2
1.3 Contents 23

CHAPTER 2 REVIEW OF RELATED WORK 28
2.1 Overview 28
2.2 Graph Theory Approach 31
2.3 Integer Programming Approaches 33
2.4 Heuristic Approaches 36

CHAPTER 3 SCHEDULING ALGORITHMS 46
3.1 Formal Definition of the Scheduling Problem 47
3.2 Optimal Scheduling Algorithm 61
3.3 Constraint Relaxing Heuristic 73
3.4 Dynamic Priority Heuristic 79

CHAPTER 4 SCHEDULING ALGORITHM RESULTS AND ANALYSES 87
4.1 Empirical Procedure 87
4.2 Optimal Scheduler Performance 89
4.3 Comparison of Heuristics 109

CHAPTER 5 SUMMARY AND CONCLUSIONS 119
5.1 Dissertation Summary 119
5.2 Applicability of Optimal and Heuristic Schedulers . . . 122
5.3 Considerations for Future Research 124

LIST OF REFERENCES 126

LIST OF TABLES

1. Image Generation Tasks' Communication and Memory Requirements 9

2. Processor Execution Performance of Each Task 17

3. IPC for Normal and Pipeline Configuration 18

4. Scheduling Constraints Addressed by Previous Researchers . . 29

5. Sequence of Events for Example Problem 51

LIST OF FIGURES

1. Example Geometry of Perspective Image Generation 5
2. Eight Tasks of Image Generation 6
3. Multiprocessor Architecture for Image Generation 15
4. A Feasible Schedule for the Example Problem 22
5. A Better Schedule which Exploits Parallelism 24
6. Graph Theory Scheduling Approach 32
7. Kartashev's Combined Resource Diagram 42
8. Four States of a Processor 49
9. Optimal Scheduler Procedure 63
10. M-ary Allocation Tree of N Tasks 65
11. Find Next Allocation Subroutine 67
12. Find Next Sequence Subroutine 70
13. Next Sequence Subroutine 72
14. Constraint Relaxing Heuristic Procedure 76
15. Modified Next Sequence Subroutine 78
16. Dynamic Priority Heuristic Procedure 83
17. GET HIGH PRIORITY Subroutine for Dynamic Priority 85
18. Random Instances Created by Random Instance Generator . . . 90
19. Optimal Scheduler Solution of Example Problem 92

LIST OF FIGURES

20. Set 1 Optimal Schedule Length Results 95
21. Set 1 Optimal Schedule Node Results 96
22. Set 2 Optimal Results - More Communication Time 97
23. Set 3 Optimal Results - Less Execution Variance 98
24. Set 4 Optimal Results - More Communication, Less Execution . 99
25. Four Communication Configurations 105
26. Communication Configuration Schedule Length Comparison . . . 108
27. Set 1 Heuristic Schedule Length Results 112
28. Set 2 Heuristic Schedule Length Results 113
29. Set 3 Heuristic Schedule Length Results 114
30. Set 4 Heuristic Schedule Length Results 115

CHAPTER 1 PRACTICAL MULTIPROCESSOR SCHEDULING

1.1 Scope

Multiprocessor systems are being considered for an increasing number of problem applications which demand large amounts of processing power. This trend is driven by the lower cost of individual processors which makes multiprocessor systems economical. However, the problem of scheduling processing tasks onto a multiprocessor system can severely limit the effective processing power of such systems. Thus, multiprocessor scheduling is becoming more important for actual systems.

The system designer must deal with the scheduling problem in a practical environment where the interaction between processing tasks can be complex. The classical work on the scheduling problem is not generally applicable because it does not consider many of the practical constraints found in real systems, such as task precedence, communication, or task deadlines. Some researchers are developing actual multiprocessor schedulers, but their ad hoc approach gives little direction for other systems.

In this dissertation, we formulate the practical multiprocessor scheduling problem in a systematic way and we develop schedulers which consider the practical constraints. We develop an optimal scheduling algorithm (with exponential time complexity) as a reference point and measure its performance, via simulation, over a variety of scheduling

problem examples. We also develop and evaluate two heuristic approaches which consider the practical constraints. Our problem formulation and scheduler investigation should provide some guidance for the designers of future multiprocessor systems and schedulers. The analysis in the results section indicates which constraints are critical and should be considered when developing a multiprocessor schedule. The results also show that our heuristic which considers different practical constraints performs better than "optimal" schedulers which do not account for practical constraints.

1.2 Problem Area and Example

Our problem area is scheduling tasks onto processors to satisfy the requirements of a given application. In Section 1.2.1 we discuss the types of applications we are concerned with and how we will represent an application as a collection of task modules with some constraints. In Section 1.2.2 we discuss multiprocessor architectures and how we will represent any architecture as a collection of processors with some constraints. Finally, in Section 1.2.3, we show how to formulate the scheduling problem in terms of the application and architecture representations.

1.2.1 Classes of Applications under Consideration

The problem of multiprocessor scheduling occurs in a variety of applications. Weather prediction, ballistic missile defense, image generation, and image processing are among those commonly identified. We are primarily concerned with these kinds of problems which require "supersystem" processing power in excess of one billion operations per second (Transactions of Computers 1982; Computer 1980). These systems achieve this processing power through tightly coupled networks of processors in a variety of intercommunication configurations. The successful use of such a system depends on properly scheduling each processor to complete its work in coordination with the rest of the system. Because of this tight coupling between processors, inefficient scheduling techniques can cause many processors to become idle and severely degrade system performance. Therefore, the scheduling problem is especially critical for these applications.

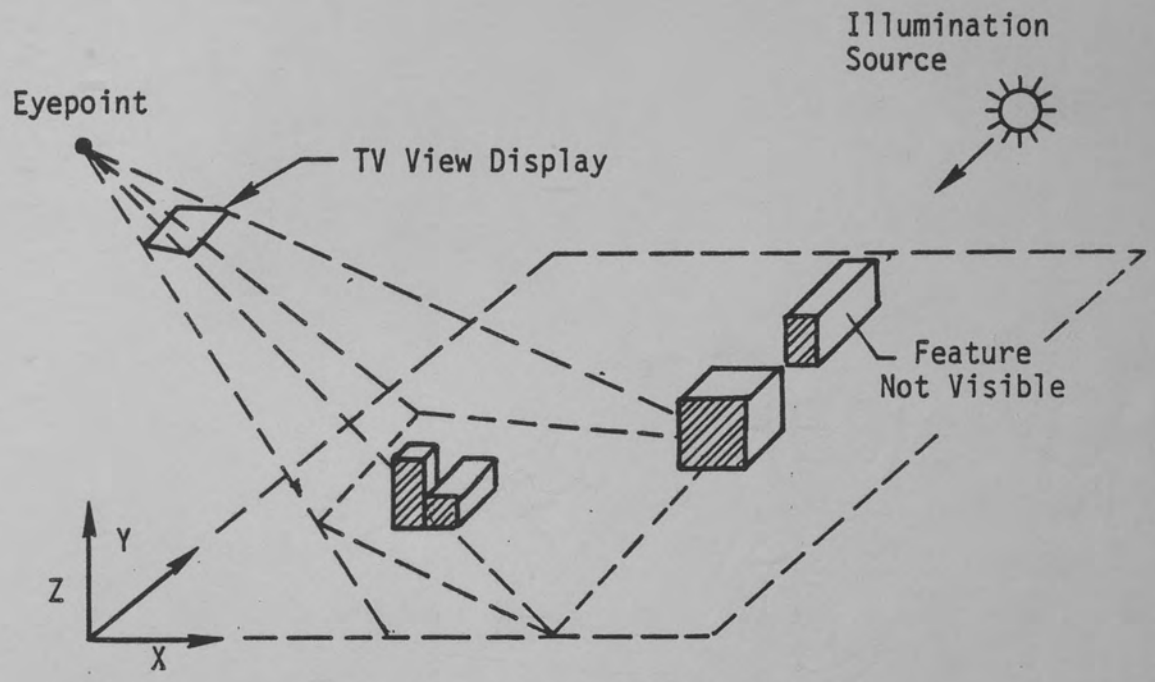
These applications are normally represented as a collection of processes or task modules. Each task requires an amount of execution time, memory, and communication with other tasks. Precedence relations and deadlines govern the period during which the task must complete its processing.

We are concerned with deterministic scheduling in which the application has already been divided into a set of tasks and all of the task constraints (i.e., execution requirement, precedence, etc.) can be determined a priori. The assumption that this kind of information will

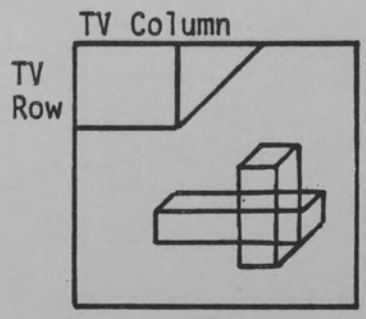
be obtainable is one reason that the class of applications is limited to supersystem-type problems. Such applications can justify the overhead costs involved in gathering this information which may require data flow analysis and test runs of the tasks. These types of applications are often scheduled deterministically in order to guarantee average and worst case behavior.

1.2.1.1 Example: Image Generation Application. We now define a simplified version of the image generation application to illustrate the constraints of the scheduling problem. We will refer to this example throughout the dissertation. The example function produces a perspective view of a data base of three-dimensional features, as shown in Figure 1. The inputs are the view window position and orientation, the sun illumination angle, and the data base features. For this example the features will be composed of planar faces where the face position is defined by the vertices of the face in Cartesian space. The output is a TV raster line display (512x512 pixels) which represents the perspective scene from the view window position. The view window and possibly the data base features can move, so a new image must be computed at a 60 hertz TV field rate (every 16 milliseconds).

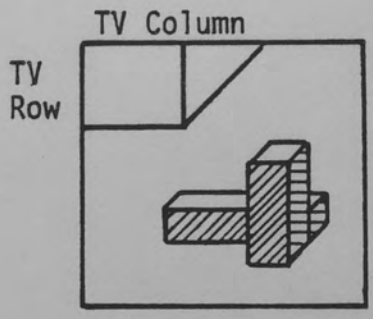
The image generation function is represented as eight tasks as shown in Figure 2. Task 1 (T1) searches through the data base to select the features which are potentially visible, as shown in Figure 1a. T2 then checks all faces of the selected features to determine which faces are potentially visible (i.e., T2 eliminates faces on the "back side"



Step A: Select Visible Features.



Step B: Project Visible Faces into TV Display Coordinates.



Step C: Color Visible Portions of Visible Features.

Figure 1. Example Geometry of Perspective Image Generation.

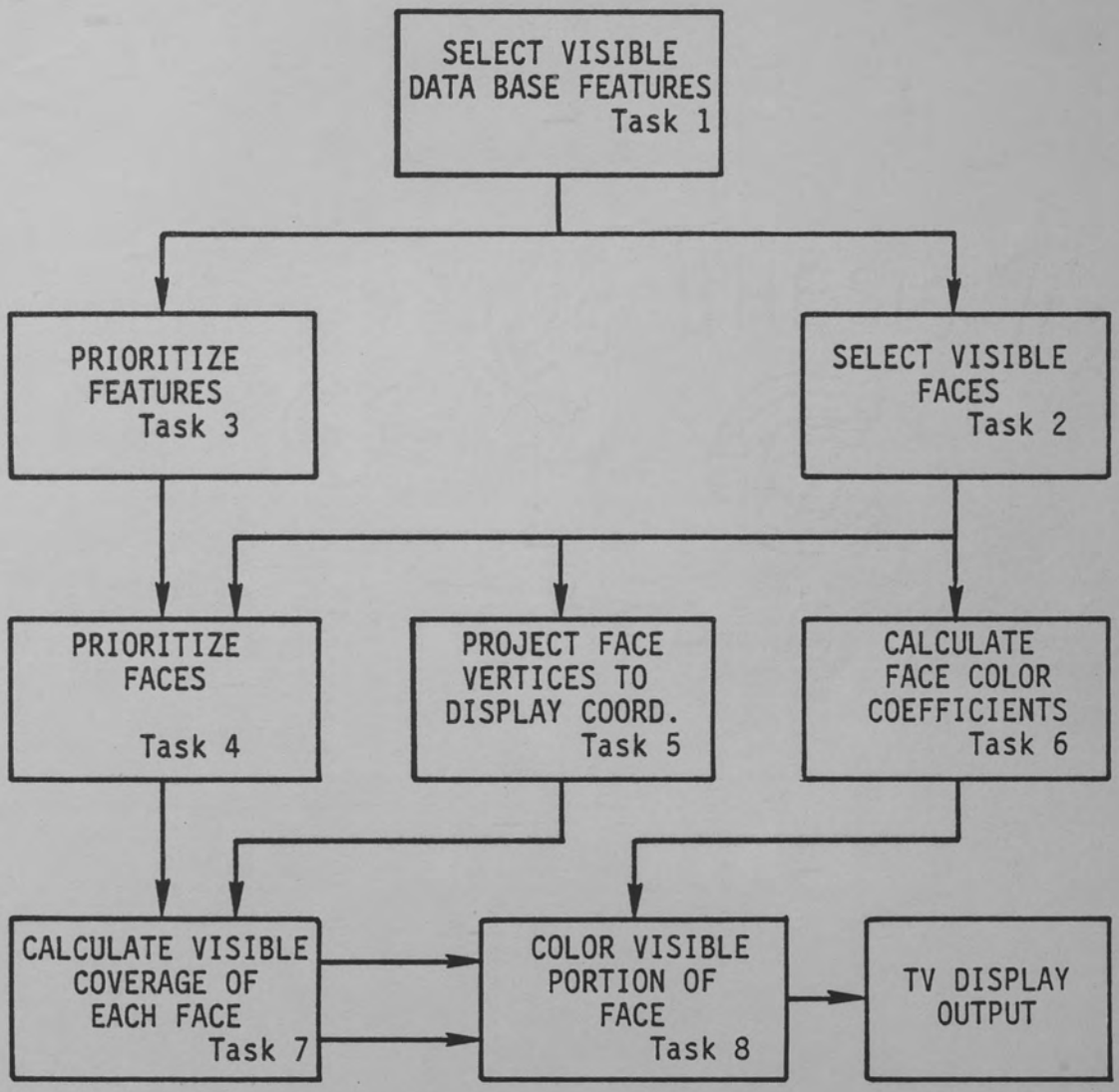


Figure 2. Eight Tasks of Image Generation.

of the feature). T3 prioritizes the features by distance so that closer features appear in front of more distant features. Figure 1b shows this where the vertical box is closer than the horizontal box and therefore the vertical box has higher priority. T4 performs a similar prioritization on the individual faces of each feature. T5 projects the face vertices from the data base coordinate system (X,Y,Z) to the display coordinate system (pixel row and column). T7 uses the face vertex positions to determine which pixels are covered by the face. T7 also resolves overlapping faces using the priority defined by T4 (e.g., in Figure 1b portions of the horizontal box overlap with the vertical box, but the faces of the vertical box have higher priority and will be used to cover those pixels). T6 calculates the color coefficients for each face. These coefficients are then used by T8 to determine the shade of color for each pixel covered by a face. The color coefficients determine the fading and shading of the face due to distance and illumination angle. The output of T8 is the color intensities (R,G,B) for each pixel in the video memory.

The eight tasks of the image generation application have precedence constraints as indicated by the directed arcs in Figure 2. For example, T3 cannot start until T1 finishes, T4 cannot start until both T3 and T2 finish, etc. We use the double lines between T7 and T8 to indicate that tasks 7 and 8 can be executed in a pipeline fashion. This is where T8 can start working on an output of T7 before T7 has finished all outputs.

Each task has an execution time constraint which is the time needed to execute the task. This is a function of both the number of processing steps to be performed and the rate of execution. Since the rate of execution can vary for different processors, we will defer defining the execution times of the example until the next section on processor architecture.

Each task has a requirement to use one or more processors concurrently. For this example, only one processor is required for each task. More than one processor could be specified for a single task when a task represents a special function which requires multiple processors. An example is a producer/consumer relationship between two processing functions. This can be modeled as a single "task" which requires two processors simultaneously.

Each task also has a deadline. The image generation function has a cycle time requirement of 16 millisec, which is represented by placing a 16 millisec external deadline on the last task, T8. Deadlines can then be propagated internally throughout the precedence tree by using the minimum execution times of each task. Other applications could have multiple external deadlines, such as when some intermediate results are required by another system at a particular time.

Another constraint on the tasks is the intertask communication requirement, or ITC. The ITC for the image generation tasks is given in Table 1a. T2 and T3 must receive 500 words from T1, T4 must receive 2000 words to T2, etc. This communication transfer will be defined to

TABLE 1

IMAGE GENERATION TASKS' COMMUNICATION AND MEMORY REQUIREMENTS

A) INTERTASK COMMUNICATION (WORDS)

FROM TASK	TO TASK 1	2	3	4	5	6	7	8
1	-	500	500	0	0	0	0	0
2		-	0	2000	2000	2000	0	0
3			-	500	0	0	0	0
4				-	0	1000	0	0
5					-	1000	0	0
6						-	0	1000
7							-	2000
8								-

B) TASK MEMORY REQUIREMENTS (WORDS)

TASK	MEMORY REQUIRED
1	1k
2	3k
3	10k
4	10k
5	10k
6	15k
7	5k
8	5k

occur after the sender completes execution and immediately before the receiver starts execution. This implies that the sender is always of higher precedence than the receiver (i.e., the sender must be executed prior to the receiver). A zero ITC is allowed between two precedence related tasks, as in the case of output dependence where both tasks output to the same data area.

The amount of communication time required is related to the number of words in the ITC and the communication rate between processors. We normally define the communication rates so that if two tasks are coresident (i.e., they execute in the same processor) then no communication time is required. This is because the two tasks share the same processor memory and have immediate access to the data to be communicated. If the tasks are not coresident then the data must be transferred by the receiving processor from the sending processor according to the available communication rate. The communication rate will be discussed in the next section on processor architecture.

The final constraint we will consider for application tasks is the task memory requirement. This is shown in Table 1b where T1 requires 1k words, etc. This requirement can reflect the memory space needed for program code and/or data storage, depending on the application and architecture. For this example the figures given include both code and data since the processors defined in the next section have a single memory for both. The sum of the memory requirements of coresident tasks cannot exceed the processor memory capacity.

1.2.1.2 Application Representation. From the previous discussion, we will represent any application in the following terms:

- An application is a collection of tasks. Each task represents a processing function, similar to the concept of a subroutine.
- The application tasks have several constraints:

  task precedence - a task cannot begin execution until all tasks of higher precedence are completed.

  task execution time - a task will require a fixed amount of time to execute on a given processor. Execution shall be nonpreemptive. The size of task execution time may vary between different processors.

  number of task processors - a task will normally require one processor for execution. If more than one processor is required, the specified number of processors must be dedicated simultaneously to the given task.

  intertask communication requirement (ITC) - the number of words which must be shared between two tasks. If tasks are not coresident then a period of communication time will be required between the processors executing the tasks.

  task memory size - the number of words which must be allocated from a processor's memory space for the task. For the set of tasks scheduled on a given processor, the sum of the task memory sizes must fit within the processor memory capacity.

  task deadline - the time limit for a task to complete execution. The time is measured from the start of the highest precedence task. The schedule length must be less than or equal to the deadline of the last task to complete execution.

This representation has intuitive appeal because these factors are considered in any system design process. As we will see in Chapter 2, however, current research in multiprocessor scheduling generally makes simplifying assumptions which eliminate some of these constraints. This representation does restrict the class of applications which will be

able to take advantage of our scheduling work. The primary restriction is that all constraints must be deterministic to allow for a deterministic scheduling. We will see that most researchers in this area make a similar assumption. However, this assumption does require that the information defining the task constraints be gathered analytically or empirically. This process can be costly and time-consuming. Thus the class of applications is narrowed to those which can afford such overhead, and supersystem-type problems generally meet this condition.

1.2.2 Computer Architectures under Consideration

There is a wide variety of computer architectures used to solve supersystem problems. Architectures are always composed of general purpose processors (e.g., a 16-bit floating point processor with a 16 k word memory), special purpose processors (e.g., a 64-point Fast Fourier Transform with a 256 k word staging memory), and communication paths between processors. Architectures can be application specific (e.g., a computer image generator), algorithm type specific (e.g., a vector processor), or general purpose (e.g., a reconfigurable architecture). We desire a model which can represent, at the system level, any type of architecture used to solve the targeted class of applications.

We represent an architecture by the performance of the individual processors on each task, the processor memory capacity, the "distance" (in time units) of communicating between each pair of processors, and the overhead time required when changing communication configurations.

These characteristics or constraints effectively define any computer system for purposes of scheduling. The execution time required by a given task can vary on different processors to differentiate between general and special purpose processors in the system. The special purpose processor will normally have excellent performance with tasks for which it was intended and arbitrarily poor performance otherwise.

The communication "distances" are specified for each pair of processors and represent the number of time units required per word during a communication between the pair of processors. The distance values can be used to represent the presence (or absence) of communication paths and the efficiencies of dedicated paths versus the penalties of shared paths. A reconfigurable architecture would have a different set of communication distances for each possible configuration. By manipulating the distance values, many different architectures can be simulated because the primary difference between pipeline, array, and vector architectures is the time required for communication.

The final architecture constraint is the configuration overhead time. This reconfiguration time is used to model the overheads of setting up a pipeline or, for the case of a reconfigurable architecture, establishing the communication paths of a new configuration.

We will generally assume that these constraints are known, which is the case when the application is to be implemented on a specific

architecture. Natural extensions can be made to develop architecture designs which would be well-suited for a given subset of problems. One example extension would be to determine the minimum number of processors needed to maintain a feasible schedule. Another extension is to investigate different communication paths (e.g., star, shared bus, cluster) to determine which type works best for a given subset of problems.

1.2.2.1 Example: Image generation architecture. The image generation application introduced in the last section is to be scheduled on the architecture shown in Figure 3. We will use this example to illustrate how we will represent an architecture in terms of processor performance, memory capacity, interprocessor communication distance and configuration change overhead.

The architecture has three processors which operate in either an independent or pipeline mode. These processors must use the inputs from the Data Memory to create the 512x512 pixel video memory image of the TV display. The 512x512 pixel video memory is also located in the Data Memory so that the TV Driver can access the data and drive the raster scan display.

The three processors are identical except for the hardware assist functions. Processor 1 (P1) and P2 are equipped with a divide function and P3 is equipped with a dot product function. Each 32-bit processor has a 64K word memory which holds the program code and working storage of all tasks to be executed on the processor. The basic execution rate

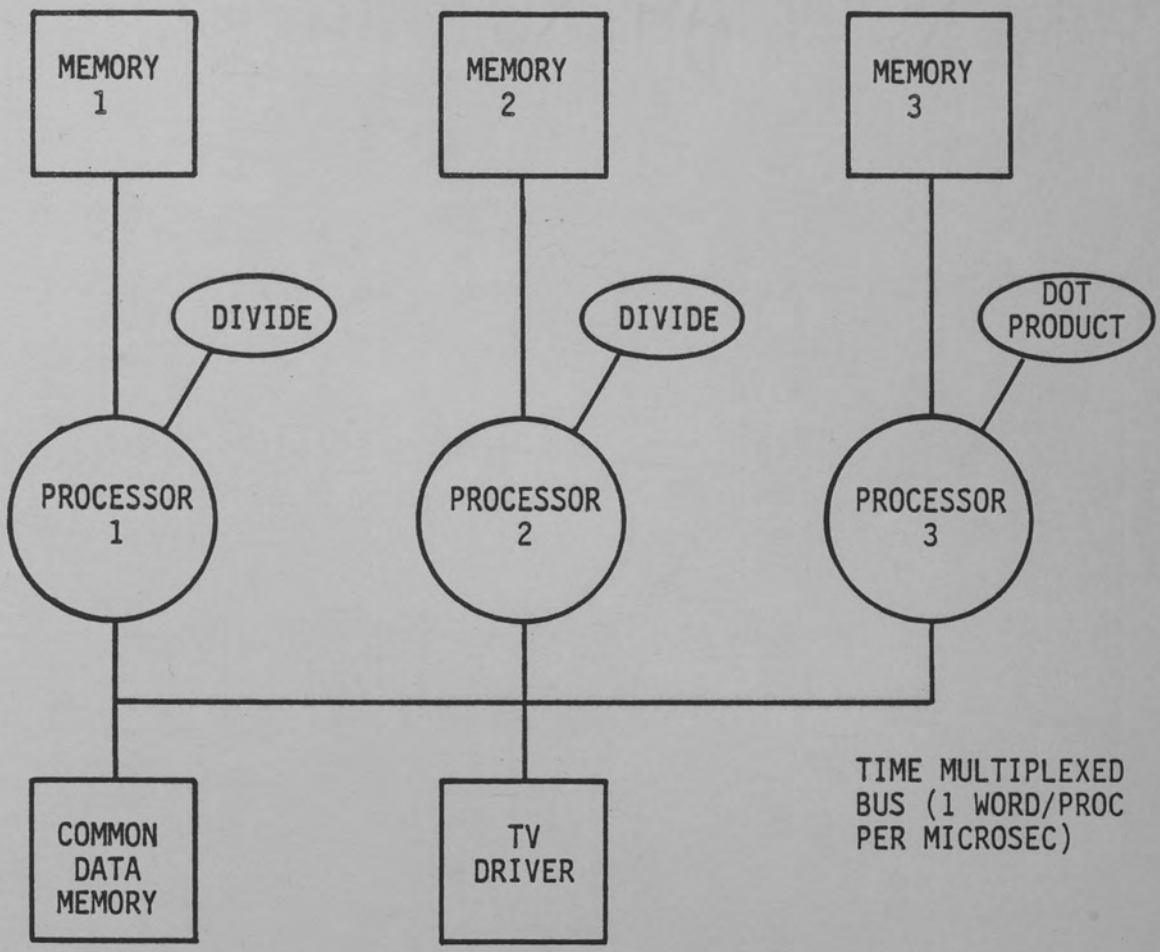


Figure 3. Multiprocessor Architecture for Image Generation.

is 5 million operations per second (MOPS) and the performance of the processors for each of the eight tasks is shown in Table 2. The difference in performance between P1 and P3 is due to the code mix of the tasks with respect to the hardware assist functions.

This performance table immediately shows that all tasks cannot execute on a single processor because the sum of execution times on any processor exceeds the deadline of 16 millisec (or 16000 microsec). Since more than one processor will be required to execute the eight tasks and the tasks have communication requirements, the interprocessor communication (IPC) time or distance becomes relevant. For the independent mode, we will assume that any processor can communicate with any other processor on the shared bus at a rate of one word every one microsec. Therefore, if X words are to be read by a task starting on P1 from a task which completed on P2, P1 must spend X microsec receiving the data from P2. The independent operation with the shared bus is shown in Table 3a by the IPC matrix where each processor is 1 microsec away from its neighbors.

As noted in the previous section, T7 and T8 can operate in a pipeline fashion where each output of T7 is allowed to be processed by T8. Table 3b shows the effective communication configuration used to implement the pipeline where the IPC has gone to zero. This reflects a configuration in which data is passed between processors over the bus during the task execution, so the time period used to transfer the block of data between T7 and T8 is not needed.

TABLE 2

PROCESSOR EXECUTION PERFORMANCE OF EACH TASK

TASK	PROCESSOR 1 (MICROSEC)	PROCESSOR 2 (MICROSEC)	PROCESSOR 3 (MICROSEC)
1	5000	5000	1500
2	1500	1500	3000
3	3000	3000	1500
4	2500	2500	7500
5	3000	3000	6000
6	500	500	3000
7	4500	4500	4500
8	4500	4500	4500
TOTAL	24500	24500	31500

TABLE 3

IPC FOR NORMAL AND PIPELINE CONFIGURATION

A) IPC (MICROSEC) FOR INDEPENDENT CONFIGURATION

FROM PROCESSOR	TO PROCESSOR 1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

B) IPC(MICROSEC) FOR PIPELINE CONFIGURATION

FROM PROCESSOR	TO PROCESSOR 1	2	3
1	0	0	0
2	0	0	0
3	0	0	0

The example system does have an overhead penalty for entering the pipeline mode. The time required to effect such a configuration change for this system will be 500 microsec. This models the time lost to achieve synchronous pipeline operation and to fill the pipeline. The scheduler must decide whether to put T7 and T8 on the same processor, on two different processors in the independent configuration, or on a set of processors in a pipeline configuration (and incur the configuration change overhead).

1.2.2.2 Architecture Representation. From the previous discussion, we will represent any architecture in the following terms:

- An architecture is a collection of processors.
- The architecture made up of processors has several constraints:

  processor performance - the performance of each processor is rated in terms of the time to execute each task. A special purpose processor will perform well with those tasks which use the special function.

  interprocessor communication (IPC) - the amount of time required to transfer one word between two processors. The IPC is defined with different values for each configuration.

  configuration change - the time overhead caused by changing the configuration, which changes the IPC.

  processor memory capacity - the amount of memory available to each processor to satisfy the task memory requirements. We will assume that all tasks are loaded into the processor memory prior to the beginning of the application run.

  Therefore the sum of the task memory requirements cannot exceed a processor's memory capacity.

This representation captures all of the architecture factors which influence scheduling. The class of architectures covered is generally

unrestricted since any architecture can be defined in these terms for scheduling purposes.

1.2.3 The Multiprocessor Scheduling Problem under Consideration

For a given application and architecture which can be represented in the terms defined in the previous sections, we wish to develop a scheduling which satisfies all of the application and architecture constraints. We assume all constraints are known a priori so we can define a deterministic schedule. The schedule is to be nonpreemptive and is established prior to the start of execution by assigning each task to run on a particular processor.

Given a schedule and the task constraints (execution time, precedence, etc.) we can compute the exact start time of each task, and, therefore, we know the schedule length. The application requirements may be such that the goal is to find any feasible scheduling, rather than an optimal feasible scheduling which minimizes the schedule length.

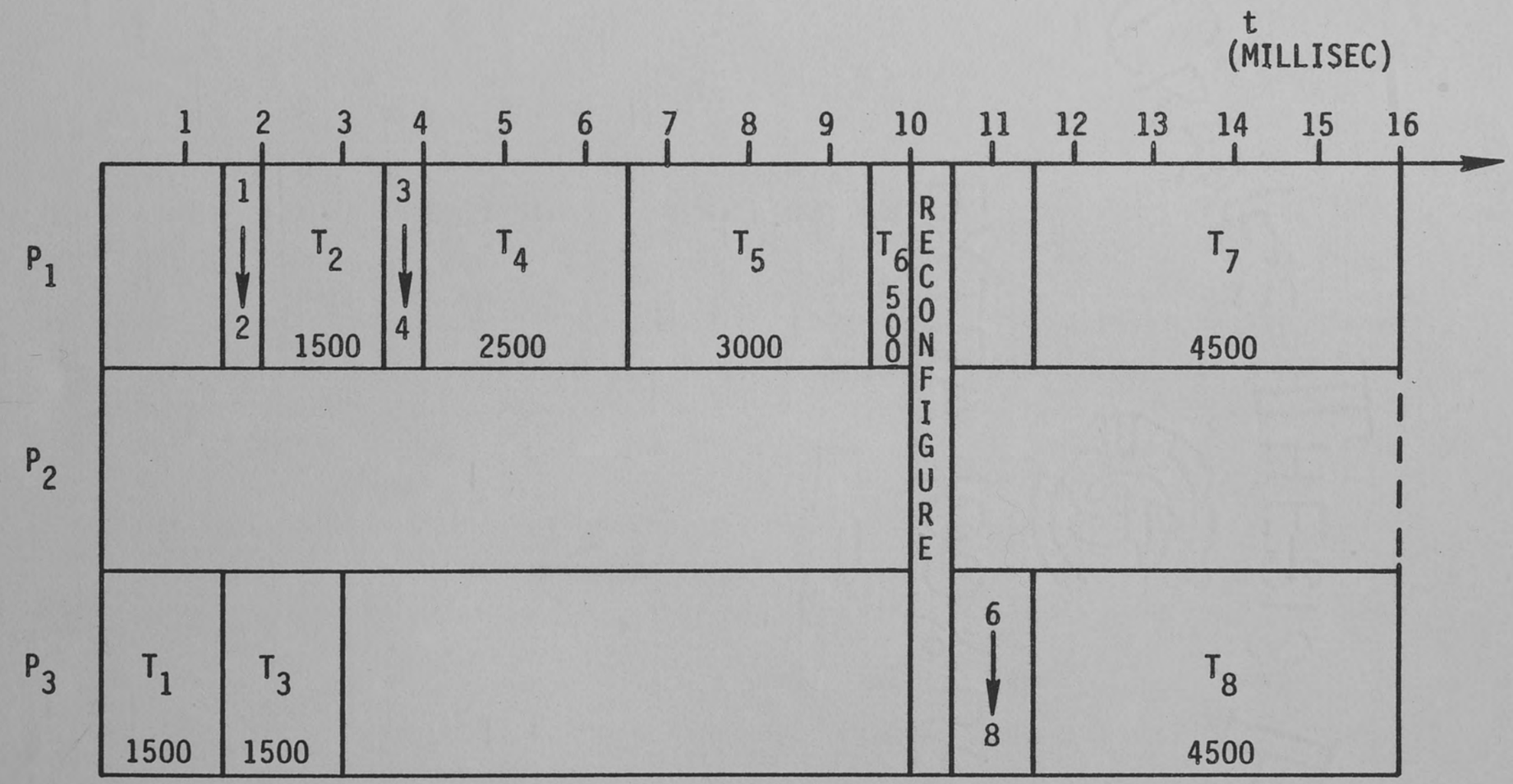
We conclude this chapter by illustrating the scheduling problem for the image generation example and then more formally defining the scheduling problem in terms of the application and architecture constraints.

1.2.3.1 Example: Image Generator Scheduling. The image generator scheduling example deals with eight tasks to be executed by a three

processor system in 16 millisec. Even this simplified problem is nontrivial and we could not guarantee an optimal solution without exercising our optimal scheduler developed later. These example schedules shown were developed manually, although the third schedule does minimize the schedule length, and is, therefore, optimal.

The simplest solution which minimizes communication time (to zero) is to schedule all tasks on a single processor. However this is not a feasible schedule since the execution time on any single processor is greater than 23 millisec (reference Table 2 for task execution times on each processor).

A second schedule, shown in Figure 4, was developed by scheduling tasks on those processors which have the best performance and which minimize communication time. This schedule is feasible since it finishes within 16 millisec. Examining this schedule in more detail, we see that task 1 (T1) is executed on Processor 3 (P3) to take advantage of P3's performance of 1500 microsec. Since T2 and T3 can be executed in parallel, and T2 runs faster on P1 than on P3, T2 is scheduled on P1 while T3 is placed on P3. However 500 microsec must be spent transferring data from P1 to P3. (reference Figure 2 for task precedence and Table 1 for ITC.) T4, T5 and T6 are also scheduled to run on P1 to reduce execution and communication times. The schedule concludes by changing the configuration to pipeline T7 and T8. This allows the two tasks to execute concurrently, but a 500 microsec change overhead is incurred and 1000 units of communication time is required for T8 to get data computed by T6.



KEY: T_1 = TASK 1; 1→2 = Comm. from T_1 to T_2

Figure 4. A Feasible Schedule for the Example Problem.

A third schedule is shown in Figure 5. This schedule is the shortest of the three. It has more execution time and communication time than either the first or second schedule. However it provides a better balance of running tasks on different processors to take advantage of performance, while reducing the communication overhead which does occur.

1.2.3.2 Summary of the multiprocessor scheduling problem. From the discussion in the previous sections of the applications and architectures under consideration, the scheduling problem is stated as follows:

Given a set of tasks, a set of processors, and the following constraints:

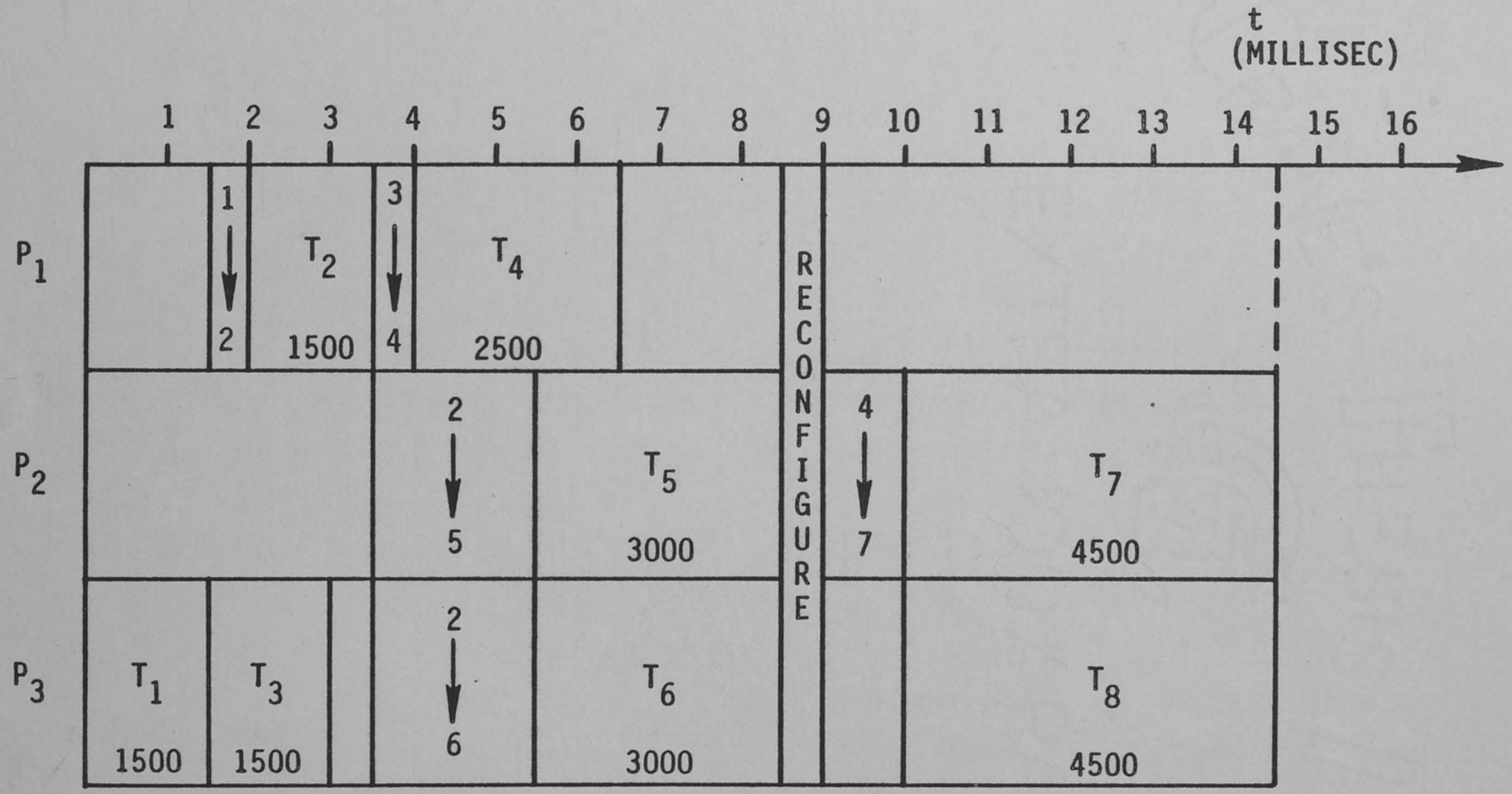
1) task execution time per processor
2) task precedence relations
3) intertask communication requirement
4) task memory requirement and processor capacity
5) task execution deadlines
6) interprocessor communication cost
7) number of coprocessors required per task
8) configuration change overhead

a.) Find a feasible schedule of the tasks on the processors, where a feasible schedule assigns each task to a processor, assigns at most one task to a processor at a time, and satisfies all constraints.

b.) Find an optimal feasible schedule which minimizes the schedule length.

1.3 Contents

The remaining chapters provide the background for this problem and describe the work which was performed. Chapter 2 reviews the related



KEY: T_1 = TASK 1

1→2 = COMMUNICATION OF T_1 TO T_2

Figure 5. A Better Schedule with Exploits Parallelism.

work to see how others have approached this problem. We show that the body of reported work has considered only subsets of the general scheduling problem that we define.

Chapter 3 contains a formal definition of the scheduling problem and describes the three algorithms which we developed to solve the scheduling problem. The first algorithm considers all of the task and processor constraints. It is optimal in that it guarantees to find the feasible schedule with the shortest schedule length, or report failure if no feasible schedule exists. However, this optimal algorithm exhibits the exponential time complexity of the NP-hard scheduling problem and is not applicable for scheduling large numbers of tasks or processors.

The second algorithm is intended to simulate other scheduling algorithms which do not consider all the scheduling constraints. This "constraint relaxing" heuristic first develops a schedule without considering one or more of the scheduling constraints. Then the true performance of the "relaxed" schedule is computed by applying the relaxed schedule to the real problem, i.e., with all of the scheduling constraints. This algorithm is based on the optimal algorithm so that the relaxed schedule is "optimal" (for the problem with the relaxed constraint). However, the true performance is generally not optimal because some of the constraints had been ignored when creating the schedule.

The third algorithm is the dynamic priority heuristic which considers all the practical scheduling constraints. The heuristic is based on priority list scheduling. The priorities are dynamically computed to guide the scheduler toward the "right" scheduling choices. This dynamic priority heuristic offers the polynomial time complexity needed for scheduling large numbers of tasks and processors.

Chapter 4 discusses the performance of these three algorithms. A problem generator is described which automatically creates scheduling problems to be solved. The optimal algorithm is evaluated for a variety of scheduling problems. The results indicate the problem sizes which can be solved using the optimal scheduler and also characterize the relationship between schedule constraints and optimal schedule length.

The constraint relaxing algorithm is evaluated to measure the performance of schedules which do not consider all practical constraints. The constraints of task precedence, communication delay, and variable task execution times are each relaxed. The schedule lengths are compared to the true optimal schedule lengths to quantify the effectiveness of other researchers' approaches when applied to scheduling problems with practical constraints.

The dynamic priority heuristic is measured against the previous two to determine how well it solves the multiprocessor scheduling problem. Although this heuristic is quite simple, it performs well because it considers the practical constraints. The performance of the dynamic priority heuristic is better than the constraint relaxing

algorithm over a variety of scheduling problems. While the heuristic could be improved upon for a given application, it verifies that successful schedulers must consider the practical scheduling constraints in a systematic way.

Chapter 5 concludes this work with a discussion of the key characteristics of the scheduling problem and algorithms. We also suggest some direction for future work in this area of multiprocessor scheduling with practical constraints.

CHAPTER 2 REVIEW OF RELATED WORK

2.1 Overview

This chapter reviews related research to show how others have attacked this problem of scheduling multiprocessor systems. Previous authors have provided a tutorial and bibliography of research approaches in this area, for example Chu (1980). Our primary concern is the types of constraints the different research approaches have considered. In particular, we will show that researchers have generally considered either precedence or communication constraints, but not both. We begin by a summary of how the previous work relates to our problem of multiprocessor scheduling with the practical constraints introduced in Chapter 1. We then provide an overview of representative work in each of three approaches to the scheduling problem:

- o Graph Theory
- o Integer Programming
- o Heuristics

Other approaches, such as analytical models (e.g., queueing theory) are not relevant because they do not consider communication or precedence constraints between tasks.

Table 4 shows how the reviewed work relates to our proposed research. Each previous work is summarized according to how the work dealt with the eight scheduling constraints listed in 1.2.3.2.

TABLE 4

SCHEDULING CONTRAINTS ADDRESSED BY PREVIOUS RESEARCHERS

SCHEDULE CONSTRAINTS	PREVIOUS RESEARCHERS (section reviewed)							
	GRAPH THEORY (2.2)	INTEGER PROGRAMMING (2.3)	INT PROG WITH HEURISTIC (2.3)	PRACTICAL SCHEDULER (2.4.1)	LOAD BALANCING (2.4.2)	AUTO-DESIGN TASK ARCHITECT. (2.4.3)	DYNAMIC ARCHITEC-TURE (2.4.4)	HI-SPEED MULTI-PROCESSOR (2.4.4)
EXECUTE TIME	OPT	OPT	OPT	HEUR	HEUR	HEUR	HEUR	HEUR
PRECEDENCE				HEUR		HEUR		HEUR
TASK COMMUN.	OPT	OPT	OPT		HEUR	HEUR	HEUR	HEUR
TASK MEMORY		OPT	OPT			HEUR	HEUR	
DEADLINES			HEUR	HEUR			HEUR	
COMMUN. DISTANCE	OPT	OPT	OPT		HEUR	HEUR		
# PROC PER TASK							HEUR	HEUR
CONFIG. OVERHEAD							HEUR	HEUR

KEY: OPT - Researcher considered constraint using optimal approach.
HEUR - Researcher considered constraint using heuristic approach.

The graph theory approach attempts to allocate the tasks onto processors by minimizing the execution and communication required using graph partitioning. This approach assumes all tasks are independent, so the task precedence constraint is not considered. This approach also does not consider the actual sequencing of the tasks on the processors, so is unable to consider deadline constraints or reconfiguration.

The integer programming approach deals with the classic task scheduling problem, with complications such as interprocessor communication and task memory. As with the graph theory approach, the integer programming formulation develops a partitioning of tasks onto processors in order to minimize the execution and communication required. This approach can consider interprocessor communication distances and memory constraints. However, it does not consider precedence or other sequence-related constraints.

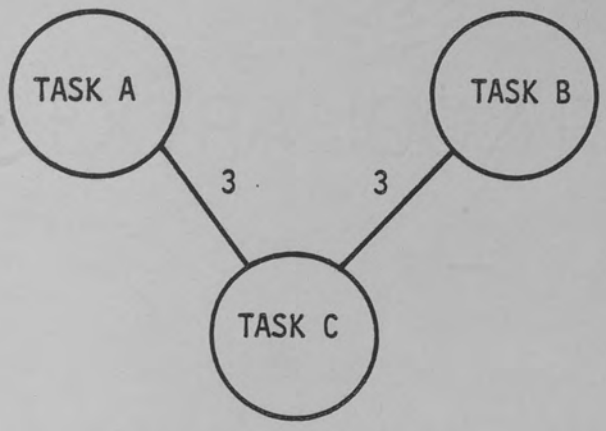
The heuristic group of papers deal with a larger set of the scheduling constraints. One paper describes good heuristics for solving the scheduling problem with precedence constraints. Two of the papers discuss how to schedule tasks onto a general multiprocessor system with the communication constraint. The last two papers discuss how to execute a given set of algorithms on reconfigurable architectures. Between the five papers, all of our practical constraints are addressed in some fashion. However none of the papers address all of the constraints in a systematic fashion.

Our own work, defined in Chapter 3, investigates optimal and heuristic algorithms which consider all constraints. Note that none of the related work covers all of our constraints, and that the previous work with optimal schedules covers only a small subset. Our work, which considers all of the constraints in a systematic fashion, will be discussed in the next chapters. The rest of this chapter briefly reviews representative works in each of the three areas of previous research to identify the constraints addressed by the previous researchers.

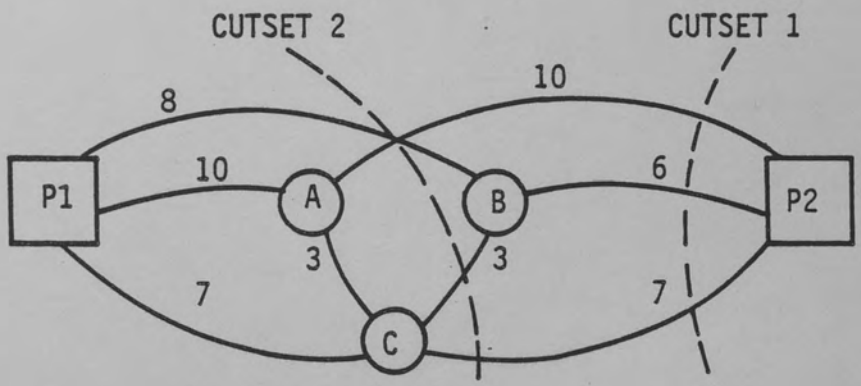
2.2 Graph Theory Approach

This approach selects a task allocation which produces a minimal cutset in a network flow graph (Stone 1977; 1978). The network flow graph represents the execution and communication costs of "flow requirement" as weighted edges connecting processors and tasks. Figure 6a shows three tasks, A, B, and C, where A and B both have a communication requirement with C. Figure 6b shows the addition to the graph of two processor nodes, P1 and P2. The weighted edges connecting task nodes with processor nodes specify the task execution time on the other processor. Therefore, an execution requirement of 8 for task B on processor P2 is represented by an edge from B to P1 with weight 8.

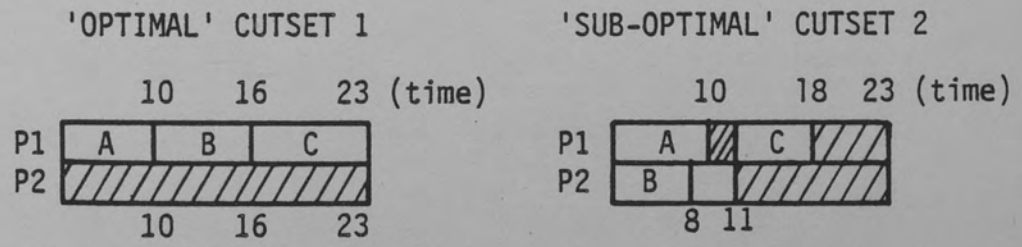
The minimal cutset (shown as 1 in Figure 6b) partitions the tasks onto the processors contained in the cutset. In this case, all of the tasks would be assigned to P1 with a total cost of 23.



a) Process Communication Requirements.



b) Process Communication and Execution Requirements.
Two Cutsets are shown in dashed lines.



c) Sub-optimal Cutset Produces Shorter Schedule Length because of Concurrency.

Figure 6. Graph Theory Scheduling Approach.

This approach has serious drawbacks. The flow graph does not include precedence relationships between tasks to model the delay of a task waiting for another task. This approach also does not minimize the schedule length, or time to complete all processors. Figure 6c shows a "nonoptimal" cutset which reduces schedule time by increasing concurrency.

2.3 Integer Programming Approaches

The research using this approach generally assumes a known multiple instruction, multiple data (MIMD) architecture and a set of tasks to be scheduled (allocated) onto the architecture. The problem is to allocate tasks onto processors to minimize the schedule time. The approaches develop an allocation by weighing requirements for task execution, intertask communication and processor load balancing.

The problem of task allocation, or task scheduling, has been investigated for over 20 years and the general problem is NP-hard (Coffman 1973). Thus, the work has concentrated on solving sub-problems (e.g., task with equal execution times or tasks in special precedence graphs) which allow a solution in polynomial time, measuring the effectiveness of heuristic methods (e.g., largest processing time first for independent tasks, list scheduling), or the effect of allowing preemption or processor sharing.

Practical work in this area followed the development of multiple processor systems for distributed information processing (Chen 1980;

Chu 1980) and for tightly coupled multiprocessor system (Efe 1982; Ma 1984). This work considers both processor execution time and interprocessor communication (IPC) time because communication can become the bottleneck in a real system.

This approach chooses a task allocation which minimizes a cost objective function. The cost objective function includes execution time and communication time, along with other application unique parameters such as storage cost for information systems. The objective function is then minimized using a branch and bound technique (BB).

Chen (1980) used this approach to design a distributed information system for a banking system. The input specification defined four cities as nodes which generated transactions, the transaction traffic, transaction processing and data base requirements, etc. Chen's integer programming model used BB to optimize an objective function with nine cost components (execution, storage, data base update, etc.) and eight constraints (communication line capacity, existence of data base, existence of a tasks on a computer, etc.). The solution output defined the optimal configuration of communication lines between cities, existence of computer and/or data base at cities, and the capacity of the system components.

Ma (1981; 1982; 1984) used the BB integer programming technique to allocate tasks to a distributed computing system. The inputs are a known MIMD system, a set of tasks, the execution requirement of each task, and the amount of intertask communication. The cost function, F,

is a summation over the task execution times and the intertask communication. The objective is to find an allocation of tasks onto processors which minimizes the sum of the execution times and the communication times. This approach considers variable task execution times, nonhomogeneous processors, variable task communication times and nonhomogeneous communication rates, or "costs", between processors. The constraints include:

a. the memory capacity of each processor must not be exceeded by the memory requirements of the tasks allocated to it.

b. a task preference matrix specifies which tasks can execute on each processor.

c. a task exclusive matrix specifies which tasks cannot be allocated to the same processor.

The output of the Ma's model is a task allocation which minimizes the cost objective function.

The main weakness in both these linear programming models is the exclusion of constraints on task dynamics such as precedence constraints or deadlines for tasks or task threads. As we showed with the graph theory approach, the model tends to group tasks on a few processors in order to minimize execution time and communication time. Thus, overhead is reduced at the expense of reducing concurrency. Ma attempts to compensate by introducing preference and exclusion matrices which force concurrency despite higher communication cost.

Unfortunately, these matrices must be manually created which effectively requires part of the allocation to be specified manually, using ad hoc criteria.

2.4 Heuristic Approaches

In this section we examine five heuristics for the general multiprocessor scheduling problem. These heuristics consider at least task precedence or task communication in developing a task allocation.

2.4.1 Critical Path Extension Heuristic

Kasahara (1984) proposes an extension to the critical path heuristic called CP/MISF (critical path/most immediate successors first). This heuristic uses a list scheduling approach with the task priorities computed based on a critical path determination. If two or more tasks have the same critical path priority, a further prioritization is made based on the number of immediate successors (descendants). A task with more immediate successors is given higher priority. This heuristic is evaluated and the worst case error (i.e. the percentage over optimal length for the heuristic schedule length) is shown to be better than the standard critical path error. The average performance is also evaluated and shown to be in the range of 5% longer than optimal. Kasahara then develops a better heuristic scheduler by using the CP/MISF in a heuristic tree search algorithm (branch and bound type).

This approach is effective for the scheduling problem with precedence and execution time constraints only. However many constraints, such as communication time and nonhomogeneous processors, are not addressed by Kasahara. This work is well supported and our own research approach described in Chapters 3 and 4 uses similar evaluation techniques.

2.4.2 Load Balancing Heuristics

The heuristic methods of Efe (1982) and Stankovic (1985) choose a task allocation by trading the communication cost against the execution load balancing (i.e., the execution load of each processor). Efe proposes a deterministic scheduler which computes the schedule before task execution begins. Stankovic proposes a realtime scheduler which accepts random task arrivals and schedules the tasks onto available processors. Both techniques consider the same set of constraints as discussed later. Efe's approach is reviewed here.

A two-stage heuristic iterates until a "sufficient" solution is found. The first stage clusters the tasks to reduce intertask communication. The second stage reassigns certain tasks from overloaded processors to underloaded processors. The resulting allocation of tasks strikes a balance between the communication and processor load balancing.

The first stage, called the task clustering algorithm, is a heuristic which assigns tasks to processors so that intertask communication is reduced. A local search technique is used which

iteratively clusters tasks with the most intertask communication. When the number of clusters will fit on the available processors, the clusters are assigned accordingly. Some provision is made for reserving certain processors for special tasks (similar to the preference matrix of Ma discussed in 2.3).

The second stage evaluates the load balancing by comparing each processor load to the theoretical average determined by the total serial task execution time and the number of processors. The processors which have acceptable loads are removed from the allocation problem along with the tasks assigned to those processors. The underloaded and overloaded processors will then be adjusted to get closer to the theoretical average.

A new problem is defined which consists of the underloaded processors, overloaded processors, task clusters from the underloaded processors, and individual tasks from the overloaded processors. The communication costs between an "underloaded cluster" and an "overloaded task" are then increased to encourage the migration of tasks to the underloaded processor. The size of the communication increase is proportional to the load difference between the processors. The new (hopefully reduced) problem is then used for another iteration of the heuristic. The heuristic terminates when all processors are acceptably balanced or the same assignment is found by two successive passes. The heuristic may not terminate.

The weakness of Efe's approach is that the model does not provide for delays from precedence constraints and communication. Also, the authors do not support the heuristic approaches by either theoretical analysis or empirical data. Stankovic's model is better supported and does provide for communication delays, however precedence constraints are also not considered.

2.4.3 Automated Design of Task-specific Architectures

Ward (1982) proposes a procedure for automatically designing a special purpose architecture which can execute a particular set of algorithms. The target applications are those where the high frequency of execution and the high speed requirements justify a special purpose machine. The goal is to automate the initial design process, and no attempt is made to produce machines capable of adapting to different algorithms.

The four steps in Ward's approach are:

1. Extract parallel tasks from sequential programs and determine firing conditions.

2. Allocate tasks to processors to meet time requirement.

3. Specify architecture using components from knowledge base.

4. Compile and load tasks into architecture.

The tasks are assigned to processors to maximize parallelism, i.e., so no two tasks on the same processor are ready for execution at the same time. Then the number of processors is reduced to minimize the system size and to reduce interprocessor communication. After the final assignment of tasks to processors, the architectural requirements such as memory size, processing power, and interprocessor communication are established. From this estimate, a knowledge base of architectural components is referenced to select processors and communication links.

The final step is to compile and load the tasks and their execution order. The operation of the architecture is similar to a data flow machine. A task is enabled and ready to execute when all predecessor tasks have executed. The task then executes and, when finished, enables its successors or descendants. The author does not report on the effectiveness of this technique.

2.4.4 Reconfigurable Architecture Heuristics

A class of architectures is being developed called reconfigurable or dynamic architectures. "Reconfigurable" refers to the ability of a multiprocessor system to change the way subsets of processors communicate and interact. These architectures are of special interest because the researchers who develop the architectures are forced to consider the scheduling or mapping of tasks onto their architectures in order to justify the reconfiguration capability.

We are interested in architectures which reconfigure in order to improve the performance of the active algorithms (or tasks). We are not interested in reconfigurable system for improving reliability. We also do not include systems such as ETH's Empress (Buehrer 1982) which is a multiprocessor machine, but which does not allow for different configurations, such as pipeline or SIMD. We shall review the works of researchers who propose reconfigurable architectures and who deal with the problem of how to prepare algorithms to be executed on their architectures. We consider two reconfigurable architectures, proposed by Kuck (1978) and Kartashev (1982).

Kartashev's reconfigurable architecture is called the Dynamic Computer (DC) (Vick 1980; Kartashev 1981; 1982a; 1982b). The problem of mapping an application onto the DC architecture is dealt with in two steps. The first step is to decompose the application into tasks or programs and measure the program resource. This is done using a P-resource (program resource) diagram which shows the memory requirement of the program and the required word width (in bits) for each major program phase or interval. The diagram also shows the execution time requirement of each interval.

The second step is to fit the P-resource diagrams of all the programs into a combined schedule or combined resource diagram. This is done using a first-fit, priority heuristic. The combined resource diagram also indicates the changes in the reconfigurable communication bus which are needed to effect different word width computers. Figure 7 is an example of the combined resource diagram and shows the fit of ten

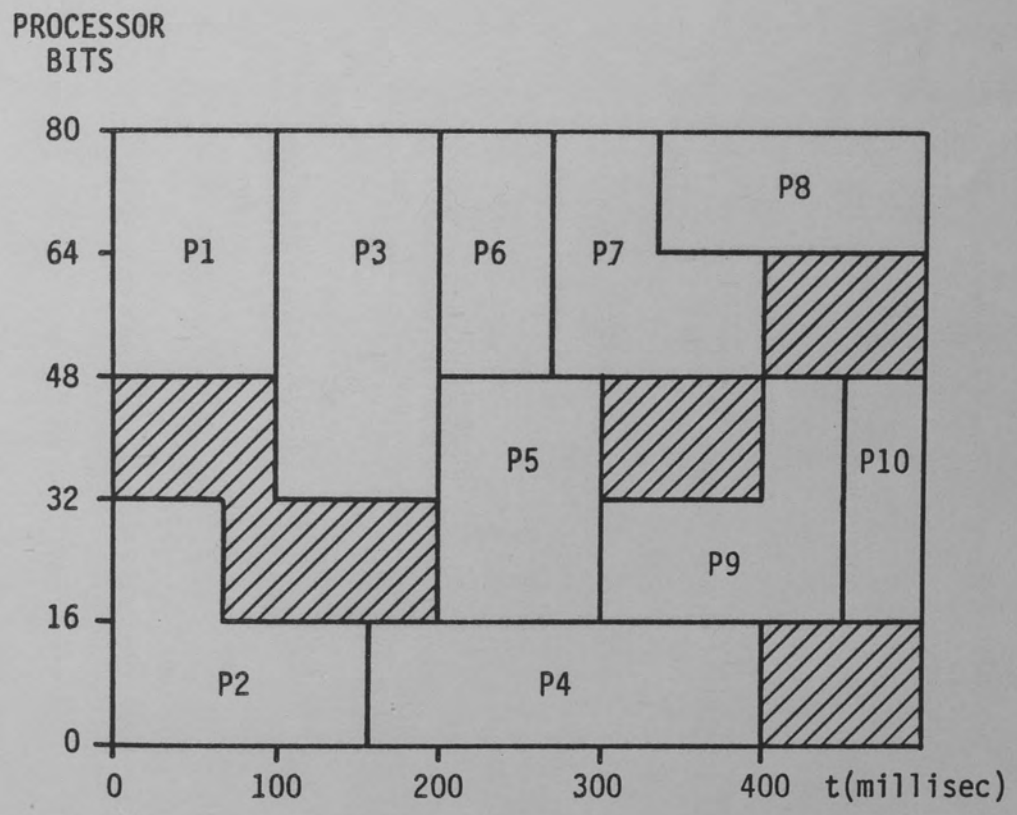


Figure 7. Kartashev's Combined Resource Diagram.

different programs (P1,P2,...,P10) onto a set of processors. Each processor is 16-bits, so the system shown in Figure 7 has five processors (80 bits). The bits required by each program defines which processors will execute the program in whole or in part.

The collection of Kartashevs' work is fairly complete, from the architecture description to the procedure for mapping programs onto the architecture. However, the concentration in developing the schedule is on fitting different word width computers together, rather than using the DC in its various modes: pipeline, master/slave, etc. Also, the performance of the heuristic for performing the schedule is not measured or evaluated by the author.

Kuck's architecture is called simply "a high-speed multiprocessor" (Kuck 1979; Padua 1980). The system is composed of multiple processor clusters (PCs) connected by a global alignment network and a global shared memory. Each PC can operate independently, can synchronize with other PCs via the global network, or can operate as a slave, with some other PCs, under control of a global control unit. Processors within a PC can operate independently, synchronized with other processors through the local network, or as a slave under control of the array control unit. Each processor has program and data memory.

This architecture can operate as an SEA (Single Execution on an Array of Data) by forcing all processors to execute the same instruction on data in their local memory. It can operate as an MEA (Multiple Execution, Array) by dividing into multiple SEAs - either to

perform multiple pipelined operations on the same array or to concurrently process multiple arrays. It can also operate as a MES (Multiple Execution, Scalar) which is a data flow type machine (Empress operates in MES mode exclusively). Reference (Kuck 1978) for further detail on Kuck's machine taxonomy.

Kuck's approach to mapping an algorithm onto the architecture has three steps. The first step is to convert the algorithm to a DAG (Directed Acyclic Graph) of Pi-blocks where a Pi-block is a simple computational node. The Pi-block is a statement or small group of statements which are "strongly connected," i.e., the data dependence between statements is cyclic. Practically, this means that the statements in a Pi-block have to be executed sequentially to ensure determinancy. Since all cycle dependencies are in Pi-blocks, any algorithm can be represented as a DAG of Pi-blocks.

The second step is to analyze the dependency of Pi-blocks which are within iteration control constructs (i.e., DO FOR loops) to increase parallelism. The techniques include rearranging the loop control structures, identifying potential concurrency within a loop, and "pipelining." Pipelining breaks a loop into smaller loops which are chained together (i.e., the ith iteration of loop j cannot start until the ith iteration of loop j-1 has completed). An evaluation is also made to determine whether the pipeline approach will be dominated by bottlenecks, where most processors in the pipeline are idle because of unequal Pi-block execution times.

The third step is to assign Pi-blocks to processors. This is similar to the task allocation in a distributed computer system problem as discussed earlier. Kuck does not add to this body of knowledge; he does note that the problem is NP-complete and that it is a common problem in scheduling theory.

The lack of discussion on the multiprocessor scheduling problem by Kuck is indicative of the need for a systematic investigation of the multiprocessor scheduling for practical systems such as Kuck's high speed multiprocessor.

CHAPTER 3 SCHEDULING ALGORITHMS

As shown in Chapter 2, the previous work in this area has developed optimal algorithms for only a subset of constraints. We also reviewed some heuristic approaches which do consider a more complete set of constraints, yet these heuristics cannot be properly evaluated since there is no comparable optimal algorithm.

In this chapter we develop an optimal algorithm and heuristic algorithms to solve the multiprocessor scheduling problem. We begin with a formal definition of the scheduling problem in terms of the constraints discussed in Chapter 1. We then describe the optimal algorithm and sketch the procedures which are used to implement the algorithm. The optimal algorithm has exponential time complexity and we discuss the theoretical worst case complexity. We then describe the constraint relaxing heuristic which is used to evaluate the performance of the other researchers' scheduling approaches. Finally, we introduce the dynamic priority scheduling heuristic which considers the key practical constraints when developing the multiprocessor schedule. The optimal algorithm and the two heuristics will be evaluated in Chapter 4 and used to investigate key characteristics of the scheduling problem.

3.1 Formal Definition of the Scheduling Problem

We define the scheduling problem as follows:

Given a set of tasks, a set of processors, and the following constraints:

1) task execution time per processor
2) task precedence relations
3) intertask communication requirement
4) task memory requirement
5) task execution deadlines
6) interprocessor communication cost
7) number of coprocessors required per task
8) configuration change overhead

a.) Find a feasible schedule of the tasks on the processors, where a feasible schedule assigns each task to a processor, assigns at most one task to a processor at a time, and satisfies all constraints.

b.) Find an optimal feasible schedule which minimizes the schedule length.

The processor scheduling problem can be formulated as a combination allocating/sequencing problem. In our formulation, the function to be minimized is the schedule length and the system of constraints account for the application and architecture constraints listed above.

3.1.1 Schedule and Schedule Length

Define the scheduling problem as having a set of m processors, P = (P1,P2,...,Pm), and a set of n tasks, T = (T1,...,Tn). Any schedule can be modeled as an allocation of tasks and a sequence of scheduling events. The allocation defines which tasks execute on which processors and the sequence defines the order that the tasks process on the processors. For our system, we consider several phases of the task

processing: execution, communication, and configuration. So any processor can be in one of four states: executing a task, communicating with another processor due an intertask communication requirement, reconfiguring due to a change in communication configuration, or idling. This definition is quite general since most other processor functions, such as operating system overhead, can be included as part of the task processing time. Figure 8 illustrates this state definition as applied to our example in Chapter 1.

The scheduling events will define any change between the four states listed above. Thus, a schedule, SCHED, is defined by the sequence, SEQ, and the task allocation, ALL:

$$SCHED = (SEQ, ALL)$$

where SEQ is a sequence of z events

$$SEQ = (E1, E2, \ldots Ez)$$

and ALL is an assignment of the n tasks onto processors and configurations

$$ALL = (A1, A2, \ldots An).$$

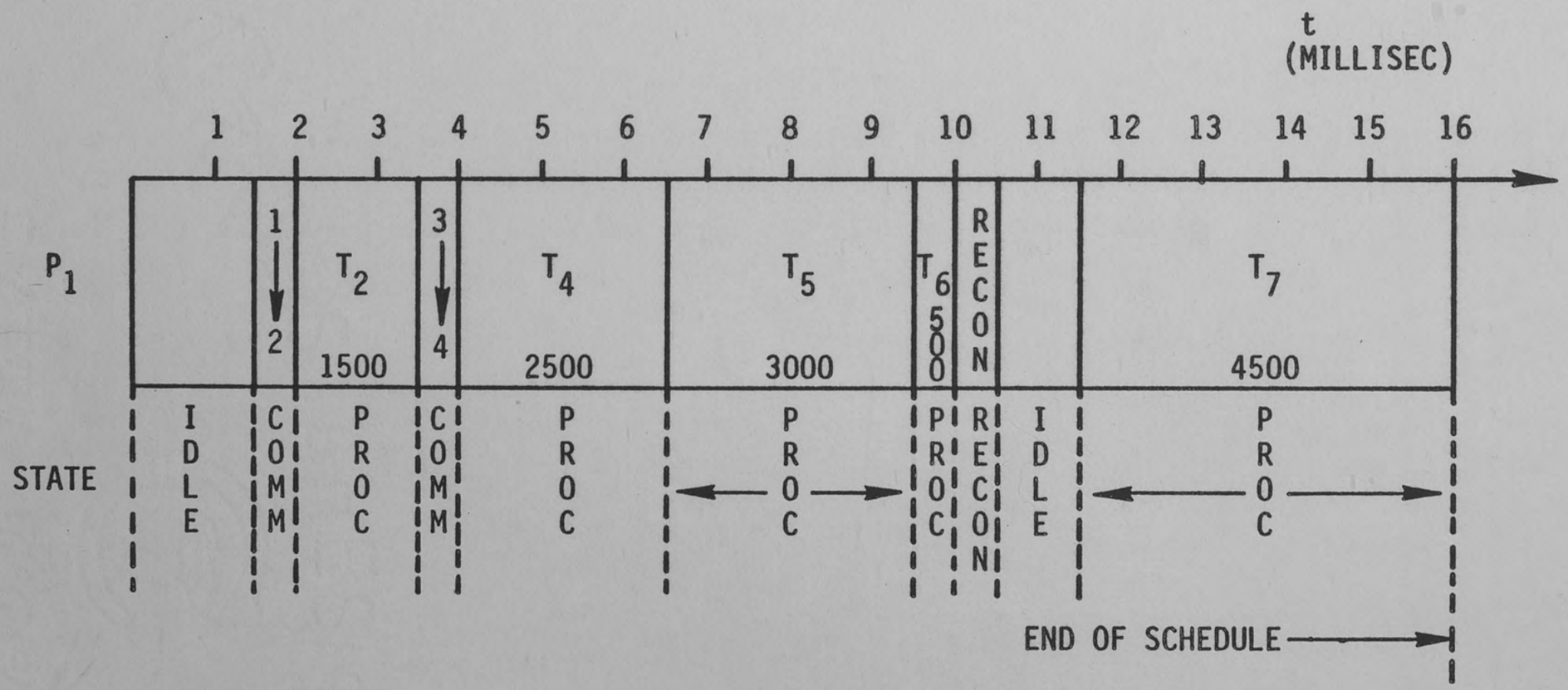
Each event, Eq, is a two tuple, Eq = (ETYPEi, ETIME), where ETIME is the time of the event and ETYPEi is one of the six task events which indicate the start or finish of one of the task processor states:

S-RFIGi = start reconfiguration required for Ti

F-RFIGi = finish " " " "

S-COMMi = start communication of Ti

F-COMMi = finish " "



FOUR STATES:

PROC - PROCESSING STATE

COMM - COMMUNICATION STATE

RECON - RECONFIGURING STATE

IDLE - IDLE STATE

Figure 8. Four States of a Processor (From Figure 4).

S-EXECi = start execution of Ti

F-EXECi = finish " "

We will use the notation t(ETYPEi) to indicate the time that ETYPEi occurred and INDEX(ETYPEi) to indicate the event sequence index of ETYPEi. In a similar fashion, we use t(Eq) to indicate the time of event q and we use TYPE(Eq) to indicate the type of event Eq. Note that the idle state is not explicitly represented but is easily computed as the absence of any other state. Each of the six event types is recorded for each task, so z = 6*n. The scheduling events for a given task will always occur in the order shown above, i.e., t(S-RFIGi) $\leq$ t(F-RFIGi) $\leq$ t(S-COMMi), etc. Multiple events can occur at the same time, such as when two tasks start execution simultaneously, i.e., t(S-EXECi) = t(S-EXECj), or when a task has no communication requirement, i.e., t(S-COMMi) = t(F-COMMi). As an example, Table 5 gives sequence events from the schedule shown in Figure 5 of Chapter 1.

Each allocation defines which processor(s), Pk, a task is assigned to and which communication configuration, R, is to be used for that task:

$$Ai = (Pk,R)$$

Normally a task requires only one processor and Pk identifies that processor. For cases where a group of coprocessors are required, we will identify the set as Pk where Pk is the first processor of the set, ordered by processor index. We will use ALLOC(Pk) to indicate the set of tasks which are assigned to Pk. The communication configuration R is a

TABLE 5

SEQUENCE OF EVENTS FOR EXAMPLE PROBLEM

EVENT - (STATE, TIME)	EVENT - (STATE, TIME)
q1 - (S-RFIG1, 0)	q25 - (S-COMM4, 3500)
q2 - (F-RFIG1, 0)	q26 - (S-COMM5, 3500)
q3 - (S-COMM1, 0)	q27 - (S-COMM6, 3500)
q4 - (F-COMM1, 0)	q28 - (F-COMM4, 4000)
q5 - (S-EXEC1, 0)	q29 - (S-EXEC4, 4000)
q6 - (F-EXEC1, 1500)	q30 - (F-COMM5, 5500)
q7 - (S-RFIG2, 1500)	q31 - (F-COMM6, 5500)
q8 - (S-RFIG3, 1500)	q32 - (S-EXEC5, 5500)
q9 - (F-RFIG2, 1500)	q33 - (S-EXEC6, 5500)
q10 - (F-RFIG3, 1500)	q34 - (F-EXEC4, 6500)
q11 - (S-COMM2, 1500)	q35 - (F-EXEC5, 8500)
q12 - (S-COMM3, 1500)	q36 - (F-EXEC6, 8500)
q13 - (F-COMM3, 1500)	q37 - (S-RFIG7, 8500)
q14 - (S-EXEC3, 1500)	q38 - (S-RFIG8, 8500)
q15 - (F-COMM2, 2000)	q39 - (F-RFIG7, 9000)
q16 - (S-EXEC2, 2000)	q40 - (F-RFIG8, 9000)
q17 - (F-EXEC3, 3000)	q41 - (S-COMM7, 9000)
q18 - (F-EXEC2, 3500)	q42 - (S-COMM8, 9000)
q19 - (S-RFIG4, 3500)	q43 - (F-COMM7, 10000)
q20 - (S-RFIG5, 3500)	q44 - (F-COMM8, 10000)
q21 - (S-RFIG6, 3500)	q45 - (S-EXEC7, 10000)
q22 - (F-RFIG4, 3500)	q46 - (S-EXEC8, 10000)
q23 - (F-RFIG5, 3500)	q47 - (F-EXEC7, 14500)
q24 - (F-RFIG6, 3500)	q48 - (F-EXEC8, 14500)

KEY TO STATES

S-RFIGi = Start Reconfig. for Task i
F-RFIGi = Finish Reconfig. for Task i
S-COMMi = Start Communication for Task i
F-COMMi = Finish Communication for Task i
S-EXECi = Start Execution for Task i
F-EXECi = Finish Execution for Task i

selection of one of the f allowable configurations. Continuing our example from Figure 5, the allocation for that schedule is given by:

```
ALL = ( A1,    A2,    A3,    A4,    A5,    A6,    A7,    A8  )
    = ( (3,1), (1,1), (3,1), (1,1), (2,1), (3,1), (2,2), (3,2))
```

The schedule is defined to start at time zero, t(E1) = 0, so the schedule length is found by the time of the last event, t(Ez). In our scheduling problem we are trying to minimize the schedule length t(Ez) while obeying all scheduling constraints. The next section defines those constraints.

3.1.2 Scheduling Constraints

The scheduling constraints serve as the rules by which a feasible schedule can be constructed and therefore serve as the rules for finding a sequence of events and an allocation of tasks. The task and processor characteristics used to define the scheduling constraints are listed below. Note the one-to-one correspondence to the application and architecture constraints discussed in 1.2.1 and 1.2.2 respectively. The application characteristics are:

Q(i,k) = task execution time of Ti on Pk.

PREC(i,j) = precedence relation between Ti and Tj
1 if Ti precedes Tj (denoted Ti <* Tj)
2 if Ti and Tj can execute as pipeline tasks (denoted Ti <*> Tj)
and 0 otherwise.

C(i,j) = number of words to be communicated from Ti to Tj

DEAD(i) = deadline time for Ti.

MREQ(i) = memory space required for Ti.

NUMP(i) = number of processors required for Ti.

D(k,l,r) = time units per communication word between Pk and Pl at communication configuration r.

MCAP(k) = memory space capacity of Pk.

REC(a,b) = overhead time to change from configuration a to b.

The constraints of the scheduling problem can now be represented using these task and processor characteristics. For the following equations we define Ti to be mapped onto Pk using configuration "a" (Ai = (Pk,a)) and Tj to be mapped onto Pl using configuration "b."

1.) Execution time constraints for all Ti

$$t(F\text{-}EXEC_i) - t(S\text{-}EXEC_i) = Q(i,k)$$

2.) Precedence constraint for all Ti

$t(S\text{-}RFIG_i) > t(F\text{-}EXEC_j)$ for all $T_j <^* T_i$. Note that
$t(S\text{-}EXEC_i) > t(F\text{-}EXEC_j)$ because $t(S\text{-}EXEC_i) > t(S\text{-}RFIG_i)$

3.) Communication constraint for all Ti

$$t(F\text{-}COMM_i) = t(S\text{-}EXEC_i)$$
$$t(F\text{-}COMM_i) - t(S\text{-}COMM_i) = \text{SUM}\,[\, C(i,j) * D(k,l,a)\,] \text{ for all } T_j <^* T_i.$$

where SUM [ ] denotes the summation of elements within the square brackets.

4.) Deadline constraint for all Ti

$$t(F\text{-}EXEC_i) \leq t(DEAD(i))$$

5.) Memory constraint for all Pk

$$MCAP(k) \geq \text{SUM}\,[\, MREQ(i)\,] \text{ for all Ti allocated to Pk}$$

6.) Processors required for Ti

a.) Exclusive use of processor(s) Pk

$t(F\text{-}EXEC_j) \leq t(S\text{-}RFIG_i)$ or
$t(S\text{-}RFIG_j) \geq t(F\text{-}EXEC_i)$ for all $T_j \neq T_i$ and Tj allocated to Pk

b.) Number of processors

SIZE (Ti) = NUMP(i) where SIZE(Ti) is the number of processors allocated for Ti

7.) Reconfiguration overhead for all Ti

t(F-RFIGi) = t(S-COMMi)
t(F-RFIGi) - t(S-RFIGi) = REC(a',a) where a' was the previous communication configuration. a' is determined by the most recent reconfiguration state for some Tj with maximum t(S-RFIGj) and with t(S-RFIGj) < t(S-RFIGi). For the very first task, a' will be set to a.

In general, these constraint definitions correspond to the intuitive descriptions offered in 1.2 where the image generation example was illustrated. Constraint 4 and 7 include two additional relationships, t(F-RFIGi) = t(S-COMMi) and t(F-COMMi) = t(S-EXECi). These constraints state that a task will immediately transition from configuring to communicating to executing without any idle time or use of the processor by another task. The rationale behind this constraint is that all of the task phases (configuring, communicating, and executing) are part of the overall task processing and our system does not permit interruptions of the task processing.

A final constraint is that the schedule will not permit all processors to be idle at the same time. Clearly, any schedule with a period of time during which all processors are idle can be improved by eliminating that period of time. Thus all reasonable schedules will not permit all of the processors to be idle.

3.1.3 Reduced Schedule Representation

The constraints listed in 3.1.2 introduce redundancy into the earlier schedule definition of 3.1.1. Some of the event times are constrained to be equal and the difference in time between many of the events are known from the task characteristics. In this section we will examine different representations of the schedule which reduce the amount of redundancy.

We can combine constraints 1, 3, and 6 to be

$$F\text{-}EXEC_i - S\text{-}RFIG_i = REC(a',a) + \sum [\, C(i,j) * D(k,l,a) \,] + Q(i,k)$$

For convenience, let F-TASKi represent the finish event for task i (F-TASKi = F-EXECi) and let S-TASKi represent the start event for task i (S-TASKi = S-RFIGi). We can formulate an equal representation of a feasible schedule SCHED by

$$SCHED' = (SEQ', ALL)$$

where SEQ' = (E1', E2', . . . En') represents the sequence of task finishes and their finish times, i.e., Eq' = (i,t) identifies the finish time for some Ti. This representation is equal in that the exact values of SCHED can be reconstructed from SCHED'. This is clearly true since, given the time of F-TASKi and the allocation, we can compute the time of the start of execution, and then the start of any communication, and finally the start of any reconfiguration. For our example:

SEQ' = ((1,1500), (3,3000), (2,3500), (4,6500), (5,8500), (6,8500),
(7,14500), (8,14500))

which is the subset of events from SEQ

(q6, q17, q18, q34, q35, q36, q47, q48)

A further reduction is possible if we are satisfied with a representation which allows us to reconstruct an equivalent feasible schedule. An equivalent feasible schedule must be feasible and must have the same schedule length. Obviously the ordering and time of internal events could be rearranged without changing the schedule length. One such rearrangement is when there is "slack" time on a processor and the task processing can be arbitrarily moved within the slack window, subject to the scheduling constraints. Another rearrangement is when the processing periods of two tasks on the same processor could be interchanged. Our reduced representation will allow only the former rearrangement because it defines the order of execution of all tasks. An equivalent schedule can be represented by

$$SCHED'' = (SEQ'', ALL)$$

where SEQ'' = (q1, q2, ... qn) is the finish order of all tasks, i.e., task q1 finishes first, task q2 finishes second, etc. Our example case would simply be SEQ'' = (1,3,2,4,5,6,7,8).

For systems which can be modeled without reconfiguration capability or overhead, we can further reduce our equivalent schedule representation by partitioning the finish order of tasks by processor, i.e., order the task finishes on each processor. This partitioning also indicates the allocation, so the reduced schedule SCHED''' can now be represented by the set of processor-partitioned sequences:

$$S''' = SEQ''' = (PSEQ1, PSEQ2, \ldots PSEQm)$$

where PSEQ1 is the set of tasks which execute on P1 ordered by their execution finish time. For our example:

$$S''' = ((2,4), (5,7), (1,3,6,8))$$

This last reduced representation of a schedule will be important for measuring the performance of schedulers which do not consider all of the scheduling constraints. Because they do not consider all of the constraints, they are unable to accurately report the start and finish times of the tasks for the schedules they produce. However, we will be able to find out the schedule length by knowing the order of task execution for each processor. Given that processor ordering, we can simulate the schedule events (with all constraints considered) and use the resulting schedule to measure the schedule length.

3.1.4 Feasible Allocation Bounds

An allocation is defined in 3.1.1 as an assignment of tasks onto processors and communication configurations. A feasible schedule requires the combination of the sequence and allocation. Our scheduling algorithms will search for a schedule incrementally, and at each step verify that no constraints have been violated. Our optimal algorithm will first try to find an allocation which has the potential to permit a feasible schedule. The allocation will then be examined for any sequences of events which produce a feasible schedule. In this section, we define those constraints which we will be able to use to identify an allocation, or subset of an allocation, which cannot render a feasible

schedule for any sequence. Obviously these will become the "bounding" tests of a branch and bound search. If a subset of an allocation is shown to violate a constraint, then all allocations containing that subset can be eliminated from consideration.

By examining the constraints listed in 3.1.2, we find that the memory constraint (constraint 5) and the number of processors per task (constraint 7b) are the only constraints independent of task sequencing. These two can therefore be used to test allocations or subsets for violations.

We can also develop a bound using the deadline constraints. Although the actual finish time of a given task cannot in general be determined during the allocation phase, we can use the precedence relations (which must be obeyed by any sequence) to determine the minimum time for the task finish. This minimum is then compared to the task deadline to check for violations. Define MINFINi to be the minimum finish time for Ti using a procedure which propagates the minimum finish time from the lowest level of the precedence tree (i.e., no antecedents) to the current task. The procedure is to find the minimum finish of the current task, Ti, allocated onto Pk is given below:

```
PROCEDURE COMPUTE.MINFIN
  MINFINi = Q(i,k)
  COMMi = 0
  DO FOR ALL Tj <* Ti
     COMMi  = COMMi + C(i,j) * D(k,l,a)
  DO FOR ALL Tj <* Ti
     MINFINi = MAX [MINFINi, (MINFINj + Q(i,k) + COMMi)]
```

This procedure depends on the existence of MINFINj, which means that all antecedents of Ti must be allocated before Ti. We will guarantee that by first ordering the tasks by pair-wise precedence (i.e., if Ti <* Tj then i < j) and then allocating the tasks in that order. If at any point we find that MINFINi > DEAD(i) then the allocation cannot lead to a feasible schedule.

Most tasks will not have an explicit deadline. For the image generator example of 1.2, only the last task, T8, had a deadline which corresponded to the 16 millisec cycle time requirement. Obviously all of the tasks could be assigned the 16 millisec deadline since they had to complete before T8. In fact, if we knew the allocation of all of the tasks, we could compute the communication and execution times and then propagate internal deadlines for all tasks. This propagation would start at the task(s) at the highest level (no descendants) and use the maximum start time for Ti to determine the deadline of all antecedents. Thus,

```
PROCEDURE PROPAGATE.DEAD
  MAXDEAD = schedule length deadline for all tasks
  DO FOR ALL Ti, i = n,n-1,...,1
     DEAD(i) = MIN [ DEAD(i), MAXDEAD ]
     DO FOR ALL ANTECEDENTS Tj,  Tj <* Ti
        DEAD(j) = MIN [DEAD(j), DEAD(i) - Q(i,k) - COMMi]
```

Unfortunately, this procedure cannot be used while an allocation is being constructed because all of the tasks must be allocated for it to work. Therefore the calculation of MINFINi above is not very useful. However, we can modify the deadline propagation procedure so that as we build the allocation in precedence order we can test MINFINi against some deadline constraints. To do this, we must make the 'best-case'

assumptions about the allocation of tasks. We use the minimum possible execution time for each task (minimum over all processors) and the minimum amount of communication time, representing these as MINQi and MINCOMMi respectively. The revised deadline propagation procedure is then:

```
PROCEDURE PROPAGATE.DEAD'
  MAXDEAD = schedule length deadline for all tasks
  DO FOR ALL Ti, i = n,n-1,...,1
     DEAD(i) = MIN [ DEAD(i), MAXDEAD ]
     DO FOR ALL ANTECEDENTS Tj,  Tj <* Ti
        DEAD(j) = MIN [DEAD(j), DEAD(i) - MINQi - MINCOMMi]
```

This can be used to check MINFINi against DEAD(i) while building an allocation. Note that MINCOMMi will normally be zero because the best-case assumption is that tasks would be coresident and not require communication. After the allocation is completed, the procedure PROPAGATE DEAD can be used to see if the allocation violates the stricter deadline constraint.

3.1.4 Feasible Sequence Bounds

We can use the constraints to define bounds while searching for sequences of a given allocation. Most of the constraints will form the rules for determining the set of possible sequences and do not have to be explicitly checked. For instance we will only consider sequencing a task when all of its antecedents have completed execution, so the precedence constraint cannot be violated. The length of execution, communication, and reconfiguration will all be computed from the characteristics so that the corresponding constraint is not violated.

The key constraint which could be violated is the deadline constraint. When examining possible sequences of a given allocation, it is best to detect a deadline constraint violation as soon as possible. The deadline calculation from procedure PROPAGATE DEAD can be used to check each task as it is scheduled. If a task violates its deadline, then no further development of that sequence is necessary.

3.2 Optimal Scheduling Algorithm

3.2.1 Scheduling Algorithm Overview

This algorithm uses a branch and bound technique to search the solution space of all possible reasonable schedules. The algorithm searches until a feasible schedule is discovered (i.e., meets all problem constraints). This feasible schedule is then recorded and the algorithm continues to search for a feasible schedule with a smaller schedule length. This process is repeated until no more feasible schedules can be found.

The last feasible schedule to be found has the minimal schedule length and is therefore optimal. If no feasible schedule is found, then none exists for the scheduling problem. The algorithm is guaranteed to find an optimal schedule because the branch and bound search will not prune a branch of possible schedules unless each of those schedules on the branch cannot be feasible. Therefore all feasible schedules are guaranteed to be inspected.

Our branch and bound algorithm is actually a two-phase process, as illustrated in Figure 9. The outer phase searches for feasible allocations using the subroutine FIND NEXT ALLOCATION. As discussed in 3.1.4, we define an infeasible allocation to be an allocation which violates a schedule constraint regardless of the sequencing. A feasible allocation is any allocation which we cannot prove to be violating a schedule constraint. For each feasible allocation found by the algorithm outer phase, the inner phase uses FIND NEXT SEQUENCE to search for all feasible schedules using that allocation. It is possible (and very likely in fact) that a feasible allocation will not render a feasible schedule. If a feasible sequence of the allocation is found, the combination is recorded as a feasible schedule.

The schedule length of that feasible schedule is then used by UPDATE TASK DEAD to define stricter task deadlines. This will have the effect of eliminating from future consideration any feasible schedules which have a longer schedule length. Once a feasible schedule is found and recorded, the inner phase continues to search for sequences of the allocation which are feasible (and must have shorter schedule length). When all sequences are exhausted, the outer phase calls FIND NEXT ALLOCATION to find another feasible allocation and the process continues. When all feasible allocations have been exhausted, the executive program terminates by reporting the most recently found (and shortest schedule length) schedule. If no feasible schedule was found, then none exists and the program reports failure.

```
procedure OPTIMAL.SCHEDULER (PROBLEM,SCHED)
;variable definition
; PROBLEM      - input definition of application and
;                architecture characteristics/constraints
; FEAS.ALL     - boolean denoting feasible allocation
; ALL          - allocation mapping
; FEAS.SEQ     - boolean denoting feasible sequence
; SEQ          - sequencing of allocation
; SCHED.FOUND - boolean denoting feasible schedule found
; SCHED        - complete schedule = (ALL,SEQ)
;subroutines called
; INIT.ALLOCATION      -  initialize allocation variables for the
;                           current scheduling problem
; FIND.NEXT.ALLOCATION -  searches forward until a new feasible
;                           allocation is found
; INIT.SEQUENCE        -  initialize sequence variables for the
;                           current allocation
; FIND.NEXT.SEQUENCE   -  searches forward to find a new feasible
;                           sequence for the current allocation
; UPDATE.TASK.DEAD     -  sets new deadline = schedule length - 1
; REPORT               -  reports the optimal feasible schedule
;                           or reports no feasible schedules exist
;
  set SCHED.FOUND = false
  call INIT.ALLOCATION
  set FEAS.ALL   = true
  do while (FEAS.ALL)           ;outer phase - get feas. alloc.
    call FIND.NEXT.ALLOCATION (ALL, FEAS.ALL)
    if (FEAS.ALL)
                        ;inner phase - get feas. seq. of alloc.
      call INIT.SEQUENCE
      do while (FEAS.SEQ)
        call FIND.NEXT.SEQUENCE(ALL,SEQ,FEAS.SEQ)
        if (FEAS.SEQ)
          set SCHED.FOUND = true
          set SCHED = (ALL,SEQ)       ;record feasible schedule
          call UPDATE.TASK.DEAD (t(SEQ(Z))
      end do              ;end inner phase
  end do                          ;end outer phase
  if (SCHED.FOUND) then call REPORT (SCHED)
  else call REPORT (false)
end procedure
```

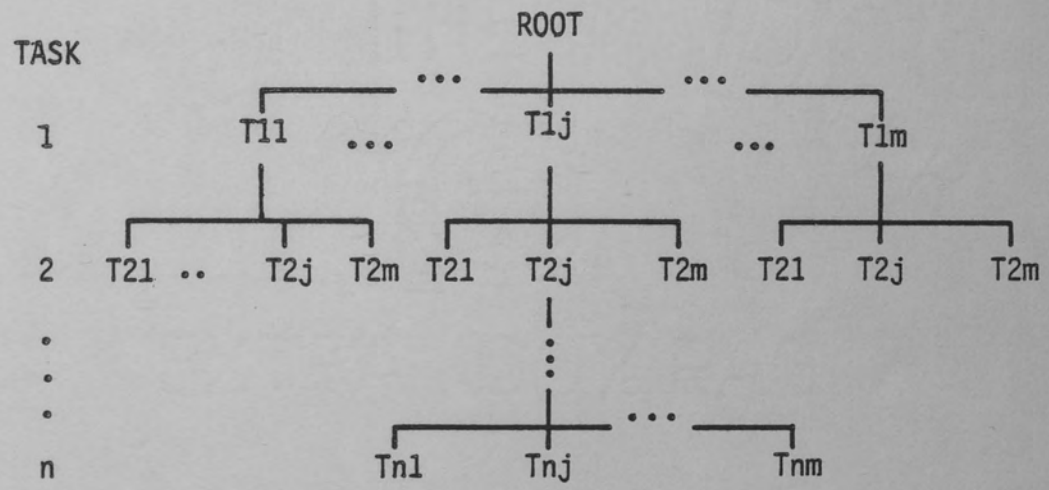
Figure 9. Optimal Scheduler Procedure.

There are many feasible allocations (up to n^m)and each feasible allocation can produce many feasible sequences (up to n|). The subroutines that the executive calls to find the next feasible allocation or the next feasible sequence are responsible for searching efficiently through the allocation and sequencing possibilities. These subroutines are discussed next.

3.2.2 Allocation Branch and Bound

Each time FIND NEXT ALLOCATION is called, it must search for a feasible allocation among the set of all possible allocations of tasks onto processors. We represent this set of possible allocations as an "allocation tree," as shown in Figure 10. The tree is structured assuming that task 1 is allocated, then task 2, etc. Each of the levels of the tree represent the different allocation choices for a specific task, given the allocations of the all the previous tasks. FIND NEXT ALLOCATION must search the tree in a methodical fashion until it finds a feasible allocation. When FIND NEXT ALLOCATION is called again, it must resume the search from the previous tree location. This search must continue until all feasible allocations are discovered. Since each of the n tasks can be allocated to any of the m processors, there are a total of n^m possible allocations. Fortunately, we can employ the scheduling constraints discussed in 3.1.4 to eliminate, or prune, parts of the tree and thus reduce our search space.

We will search the tree in a depth-first fashion. At each level the allocation choices will be evaluated and the task will be allocated to a



Note: Tij indicates that task i is assigned to processor j

Figure 10. M-ary Allocation Tree of N Tasks.

processor such that none of the allocation constraints are violated. This process continues from level to level until either all of the tasks are allocated or the task at the current level cannot be allocated without violating the allocation constraints. If the task at the current level cannot be allocated then all of the allocation possibilities below that point are ignored and the subroutine backtracks to level which has feasible allocation possibilities. The program then continues forward until it must backtrack again, or all of the tasks are successfully allocated.

If all of the tasks are successfully allocated then the feasible allocation is returned to the executive. The executive will then use FIND NEXT SEQUENCE to search for any feasible schedules using that allocation. When all sequences are exhausted the executive recalls FIND NEXT ALLOCATION which backtracks from the current task level (level n since all tasks are allocated) to level n-1 and continues to search for another feasible allocation. FIND NEXT ALLOCATION will eventually finish searching the allocation tree and will report that no additional feasible allocations exist.

The subroutine FIND NEXT ALLOCATION is given in Figure 11. The outer do while loop performs the depth first forward search, advancing form one task level to the next as long as the allocation remains feasible. The subroutine INIT ASSIGN is called when each level is entered from "above," e.g., if task 5 is to be allocated after task 4 has been allocated, then INIT ASSIGN (5) is called. INIT ASSIGN serves to initialize the status of the current node of the tree so that all

```
subroutine FIND.NEXT.ALLOCATION (ALL, FEAS.ALL)
;variable definition
;   N           - number of tasks in scheduling problem
;   FEAS.ALL    - boolean denoting feasible allocation
;   ALL         - allocation mapping
;   TASK        - index of the last task to be allocated
;                   initialized to 0 by INIT.ALLOCATION
;   NEW         - boolean value for each task which
;                   initialized true by INIT.ALLOCATION
;
;subroutines called
;   INIT.ASSIGN - prepare for first assign of a task in
;                   a forward search
;   NEXT.ASSIGN - allocate task #TASK+1.  Iff allocation is
;                   feasible, set FEAS.ALL=true
;
  do while ( (TASK .lt. N) .and. FEAS.ALL )

    if ( NEW(TASK) )
      call INIT.ASSIGN (TASK)     ;this is a forward search
      set NEW(TASK) = false     ;init task's allocation state
    call NEXT.ASSIGN (TASK, ALL, FEAS.ALL)

    do while ( not.FEAS.ALL .and. (TASK .gt. 0))
      NEW(TASK) = true ;flag this task for forward search
                       ;backtrack to previous level if infeasible
      set TASK = TASK - 1
      call NEXT.ASSIGN (TASK, ALL, FEAS.ALL)
    end do
    set TASK = TASK + 1
   end do
return
```

Figure 11. Find Next Allocation Subroutine.

allocations for that node will be considered. NEXT ASSIGN (5) is then called to perform the actual allocation using the best allocation at that level 5, where "best" is defined using the minimum execution and communication times computed for that allocation.

During a backtrack operation, a task level will be entered from "below," e.g., level 6 has no feasible options so it backtracks to level 5. At this point we call NEXT ASSIGN since we want to advance to the next best allocation at the current level, e.g., level 5. If the next best allocation is not feasible, we continue to backtrack. If it is feasible, we resume the forward search from that point.

3.2.3 Sequencing Branch and Bound

Each time FIND NEXT SEQUENCE is called, it must search for a feasible sequence among the set of all possible sequences of events for the given allocation of tasks onto processors. The structure of FIND NEXT SEQUENCE for controlling the search of sequences is similar to the control structure of FIND NEXT ALLOCATION. For this case, the search tree is the set of all possible scheduling events within the allocation. There are 2*n levels, or events, where each event is either a task start or a task finish. Again, the search is depth-first with the subroutines NEXT SEQ and BACK SEQ doing the investigation.

As with the allocation processing, we will search the sequence tree in a depth-first fashion. At each event level the sequencing choices will be evaluated and one chosen. Multiple event options will

be available only if more than one event is ready to occur at the same time, e.g., two tasks are ready to start execution at the same time. A choice between options must be made if they are mutually exclusive, e.g., the two ready tasks are allocated to the same processor. One option must be chosen and then the next event must be found. This process continues from event to event until either the last event is successfully scheduled (i.e., all tasks have started and finished) or the current event is not feasible because it violates a scheduling constraint, in particular the deadline constraint. If the event at the current level has no feasible options, then we backtrack to an event which has feasible options. The program then continues forward until it must backtrack again, or all of the events are scheduled.

If all of the events are successfully scheduled then the feasible sequence and allocation is returned to the executive. The executive will then record the feasible schedule and use the schedule length to define new, smaller task deadline values. The executive then recalls FIND NEXT SEQUENCE which backtracks from the current event level (level 2*n since each task must start and finish) to level 2n-1 and continues to search for another feasible sequence. FIND NEXT SEQUENCE will eventually finish searching the SEQUENCE tree and will report that no additional feasible sequences exist.

The subroutine FIND NEXT SEQUENCE is given in Figure 12. The outer DO WHILE loop performs the depth first forward search, advancing from one event level to the next as long as the sequence remains feasible. FIND NEXT SEQUENCE calls NEXT SEQ to move forward a single event. NEXT

```
subroutine FIND.NEXT.SEQUENCE (ALL,SEQ,FEAS.SEQ)
;variable definition
;   N            - number of tasks in scheduling problem
;   ALL          - allocation mapping
;   FEAS.SEQ     - boolean denoting feasible sequence
;   SEQ          - sequencing of allocation
;   EVENT        - index of the last event to be scheduled
;                  initialized to 0 by INIT.SEQUENCE
;   LAST.EVENT   - global boolean set true when all tasks
;                  have finished
;   BACK.START   - boolean set true by subroutine BACK.SEQ
;                  iff the last event backtracked was a task start
;
;subroutines called
;   NEXT.SEQ     - determine next event to occur.  Iff event is
;                  feasible, set FEAS.SEQ=true
;   BACK.SEQ     - backtrack event #EVENT and undo its effects
;

  LAST.EVENT = false
  do while ( not.LAST.EVENT .and. FEAS.SEQ )

    call NEXT.SEQ (EVENT, FEAS.SEQ)

    do while ( (EVENT .gt. 0) .and. (not. FEAS.SEQ) )
      call BACK.SEQ (EVENT,BACKSTART)
      set EVENT = EVENT - 1
      if (BACKSTART) then call NEXT.SEQ (EVENT,FEAS.SEQ)
    end do

    set EVENT = EVENT + 1
  end do
return
```

Figure 12. Find Next Sequence Subroutine.

SEQ considers only the precedence feasible sequences when selecting the next event in a forward search. This is accomplished by maintaining an event-based simulation of the states of all tasks. Only tasks which have their precedence relations satisfied can ever be scheduled. As shown in Figure 13, NEXT SEQ first checks all of the idle processors to see if there is a ready start event, i.e., a task ready to begin execution. If a start event is found, it is recorded and the subroutine returns to FIND NEXT SEQ. If more than one start event option is available, then the "best" one is chosen which has the smallest execution time. If no start event is found, NEXT SEQ determines which currently executing task will finish next. The simulated clock is advanced to the time of this next finish event and the finish event is recorded. This subroutine is called repeatedly in a forward search to record the next scheduling event and advance the simulation clock.

If any of the options is infeasible (i.e., a task cannot be started because it will not finish before its deadline) then none of the options need be considered and backtracking is required. (Obviously the task which violates the deadline constraint will always violate the constraint for any future scheduling.) When backtracking is required, the inner DO WHILE loop of FIND NEXT SEQ is activated to repeatedly call BACK SEQ EVENT. BACK SEQ EVENT simply reverses the effect of the previous scheduling event and reverses the clock. If the backtracking event was a start, then BACK SEQ EVENT returns a flag so that any other feasible options at that start event can be investigated. The outer loop of FIND NEXT SEQ will resume the forward search at that point.

```
subroutine NEXT.SEQ (EVENT, FEAS.SEQ)
;variable definition
;  EVENT          -   index of current event
;  FEAS.SEQ       -   boolean set false if constraint violated
;  FOUND.EVENT    -   boolean set true if an event is found
;  PROC           -   local index to check all processors
;subroutines called
;  FIND.START     -   checks if a ready task is available to be
;                     started on the processor
;  FIND.FINISH    -   called if no starts available.  Finds the
;                     next task finish and advances clock to finish

  set FEAS.SEQ = true
  set FOUND.EVENT = false

  do for PROC = 1 to M
    call FIND.START (EVENT, PROC, FOUND.EVENT, FEAS.SEQ)

  if (not FOUND.EVENT and FEAS.SEQ)
    call FIND.FINISH (EVENT, FOUND.EVENT, FEAS.SEQ)

return
```

Figure 13. Next Sequence Subroutine.

3.3 Constraint Relaxing Heuristic

The scheduling algorithm described in 3.2 is guaranteed to find the optimal schedule, but the exponential time complexity of the scheduling problem limits the algorithm to small problems. The execution time performance of the OPTIMAL SCHEDULER is reported in detail in Chapter 4. It is sufficient here to note that a problem with only sixteen tasks and three processors may require evaluating over ten million schedule nodes, representing several hours of computing time. This time would grow to days and years with small increases to the numbers of tasks or processors. Thus, in order to schedule large numbers of tasks and processors, we must relax our goal of optimality and look for heuristics which will produce a "reasonably good" schedule.

Heuristic scheduling approaches are difficult to compare without a known baseline. Our technique for comparison is to develop a benchmark set of schedules with known optimal schedules. We will then compare our heuristic to that benchmark. We also need to show how our heuristic compares to other researchers' approaches. Since their specific results are not generally available and reproducible, and since they did not evaluate their algorithms against an optimal baseline, we have developed the constraint relaxing algorithm to empirically evaluate their approaches.

The key about other researchers' approaches is that they do not consider one or more of the practical scheduling constraints, as shown in Chapter 2. We will refer to a constraint not considered as a relaxed

constraint. We can model their scheduling approach using our optimal algorithm, with the corresponding constraints relaxed, and call the resulting schedule a relaxed schedule. Our optimal algorithm will obviously produce a relaxed schedule at least as good as any other scheduler implementation. We can then measure the true length of the relaxed schedule by simulating that schedule with the actual scheduling problem with all constraints. The resulting length from the relaxed schedule is a good measure of the scheduling approach which does not consider the specific constraint. The constraints we will allow to be relaxed are communication requirements, precedence relations, and variable task execution times.

3.3.1 Constraint Relaxing Heuristic Overview

The constraint relaxing heuristic works in three steps:

1. Relax selected constraints of actual problem
2. Find schedule for relaxed problem
3. Use relaxed schedule for actual problem with all constraints reintroduced

Note that the relaxed schedule found in step 2 will provide the allocation of tasks to processors and the sequencing of tasks within a processor. Step 3 will then use the relaxed schedule to determine the actual start and finish times for each task and the actual schedule length. As noted in 3.1.3, we can evaluate the schedule length by

reconstructing the actual events given the order of the events. We will perform an event-based simulation of the task executions and use the defined order of events to resolve any conflicting event options. The resulting schedule length will be the measure for evaluating the relaxed schedule.

We find the relaxed schedule using the optimal schedule algorithm with the selected constraints neutralized. For example, the precedence constraints might be eliminated or the task executions times might be set to a constant. Another researcher's scheduling algorithm could be used here in place of our own optimal scheduling algorithm. But for comparison purposes, our optimal algorithm produces a relaxed schedule which is at least as good (short schedule length) as an algorithm from the previous research which does not consider the relaxed constraint.

The CONSTRAINT RELAXING procedure is shown in Figure 14. The executive first relaxes the selected set of constraints by calling SAVE CONSTRAIN. SAVE CONSTRAIN simply saves a copy of the problem and then neutralizes the selected set of constraints. The executive then calls the normal OPTIMAL SCHEDULER procedure to find the optimal relaxed schedule. The original constraints are restored by RESTORE CONSTRAIN and the the relaxed schedule is evaluated for the fully constrained scheduling problem. The evaluation is performed using the allocation developed for the relaxed schedule. The schedule events are determined by calling FIND NEXT SEQUENCE once. There is only one possible sequence to "search" since the sequence is defined by the relaxed schedule.

```
procedure CONSTRAINT.RELAXING (PROBLEM,SCHED)
; PROBLEM      - input definition of application and
;                architecture characteristics/constraints
; RELAX.PROBLEM - problem definition with relaxed constraint
;                 removed
; RELAX.SCHED - the optimal schedule for the realxed problem
; ALL          - allocation mapping
; FEAS.SEQ     - boolean denoting feasible sequence
; SEQ          - sequencing of allocation
; SCHED.FOUND - boolean denoting feasible schedule found
; SCHED        - complete schedule = (ALL,SEQ)
;subroutines called
; SAVE.CONSTRAIN        -  record original constraints and remove
;                          constraints to be relaxed
; OPTIMAL SCHEDULER     -  finds optimal schedule for relaxed prob.
; INIT.SEQUENCE         -  initialize sequence variables for the
;                            current allocation
; FIND.NEXT.SEQUENCE    -  searches forward to find a new feasible
;                            sequence for the current allocation
; REPORT                -  reports the optimal feasible schedule
;                            or reports no feasible schedules exist

; first find the optimal schedule for the relaxed problem

  call SAVE.CONSTRAIN (PROBLEM, RELAX.PROBLEM)
  call OPTIMAL.SCHEDULER (RELAX.PROBLEM, RELAX.SCHED)

; now evaluate the relaxed schedule on fully constrained problem

  call RESTORE.CONSTRAIN (PROBLEM, RELAX.PROBLEM)
  call SET.ALLOCATION (RELAX.SCHED, ALL)
  call INIT.SEQUENCE
  call FIND.NEXT.SEQUENCE'(ALL,SEQ,FEAS.SEQ)
  if (FEAS.SEQ)  then call REPORT (SCHED)
  else call REPORT (false)

end procedure
```

Figure 14. Constraint Relaxing Heuristic Procedure.

3.3.2 Constraint Relaxing Subroutines

Three new subroutines defined for this heuristic are READ CONSTRAIN, SAVE CONSTRAIN, and RESTORE CONSTRAIN. These subroutines simply provide the logic to determine which constraints should be relaxed, relax the selected constraints, and restore the selected constraints.

We also modify FIND NEXT SEQ to force the task execution order of the relaxed schedule to be repeated. Actually, we implement this by modifying NEXT SEQ (from the optimal scheduler, Figure 13) so that when the next start event must be in accordance with the order of the relaxed schedule. The modified version of NEXT SEQ is given in Figure 15. The only modification is that the subroutine GET HIGH PRIORITY is called before searching for start events. GET HIGH PRIORITY uses the task execution order from the relaxed schedule to control when ready tasks are allowed to begin execution. In effect, the order of the relaxed schedule becomes another precedence constraint because tasks are restricted to execute in the order of the relaxed schedule.

3.3.3 Constraint Relaxing Scheduler Time Complexity

The time complexity of this algorithm can be developed by examining the major components called by the CONSTRAIN executive:

- o READ, SAVE, RESTORE CONSTRAIN = O(n)
- o Find relaxed schedule = O(OPTIMAL SCHEDULER)
- o Evaluate relaxed schedule = O(nlogn)

```
subroutine NEXT.SEQ (EVENT, FEAS.SEQ)
;variable definition
;  EVENT          -   index of current event
;  FEAS.SEQ       -   boolean set false if constraint violated
;  FOUND.EVENT    -   boolean set true if an event is found
;  PROC           -   local index to check all processors
;subroutines called
;  FIND.START     -   checks if a ready task is available to be
;                     started on the processor
;  FIND.FINISH    -   called if no starts available.  Finds the
;                     next task finish and advances clock to finish
;  GET.HIGH.PRIORITY - get highest priority task for each proc.
;
  set FEAS.SEQ = true
  set FOUND.EVENT = false

  call GET.HIGH.PRIORITY

  do for PROC = 1 to M
    call FIND.START (EVENT, PROC, FOUND.EVENT, FEAS.SEQ)

  if (not FOUND.EVENT and FEAS.SEQ)
    call FIND.FINISH (EVENT, FOUND.EVENT, FEAS.SEQ)

return
```

Figure 15. Modified Next Sequence Subroutine.

For our case, the OPTIMAL SCHEDULER is exponential so the overall complexity is exponential. However, we could have found a nonoptimal relaxed schedule using a heuristic, such as the heuristics of previous researchers. Since almost every heuristic is at least O(nlogn), the overall complexity would be governed by the complexity of the heuristic.

3.4 Dynamic Priority Heuristic

This heuristic is based on the simple list scheduler with some modifications to dynamically adjust the priority list order. In a list scheduler, tasks are scheduled during actual application processing. Idle processors request a task for execution and the scheduler selects one of the ready tasks (tasks with all precedence relations satisfied) for that processor. The selected ready task is scheduled onto that processor for execution. When the task finishes execution, the processor becomes idle again and requests another task for execution. This is called list scheduling since the scheduler selects a ready task for an idle processor based on a schedule list which prioritizes the tasks.

This heuristic develops a schedule by simulating the operation of a list scheduler. We use the same event-based simulation used by the optimal scheduler (reference 3.2.3) and by the constraint relaxing heuristic (reference 3.3.2). For our heuristic, the event-based simulation keeps track of the start and finish of tasks. Each time a task finishes, the list scheduler will assign one of the ready tasks to the idle processor. We record the order of execution of the simulated operation and that order serves as the schedule.

This dynamic priority heuristic prioritizes the schedule list in an attempt to produce a short schedule length. The priority of each task is developed using the different constraints defined in 3.1, such as task execution, task communication, deadlines, etc. The priority is dynamic because the priority of a given task will depend on the previous scheduling activity up to the moment the task is scheduled.

We also introduce a lookahead extension which allows the scheduler to accommodate high priority tasks which are "almost ready." This mechanism allows the scheduler to anticipate that a high priority task will be ready to execute soon. The scheduler can then reserve a processor for the high priority task so that the high priority task can begin execution as soon as it becomes ready.

3.4.1 Dynamic Priority Heuristic Overview

The dynamic priority heuristic performs an event-based simulation of the tasks executing on the set of processors. The priority of each ready task is computed for every idle processor and the task with the highest priority is scheduled onto the corresponding processor. The task with the highest priority and the processor it is scheduled on are removed from the set of ready tasks and set of idle processors, respectively. The scheduling process repeats for the remaining ready tasks and idle processors until no more ready tasks or idle processors are available. The simulation then advances to the next event.

The task priority is computed as a weighted sum of factors derived from the practical constraints defined in 3.1. Some of these factors give priority to one task over another and some of these factors give priority to one processor over another for a specific task. The factors are:

- o Task execution time
  - variable per processor. Favors processors which execute the task faster (called TASK EXEC)
  - variable between tasks. Favors tasks which require longer execution (called PROC EXEC)
- o Precedence relations
  - precedence level - favors tasks at a higher level of precedence (i.e., fewer ancestors)
  - descendant degree - favors tasks with a large number of immediate descendants
- o Intertask/interprocessor communication - favors processors which reduce the task's communication requirement
- o Task execution deadline - favors tasks which have immediate deadlines
- o Task memory requirement - favors processors which have a lot of available memory

Note that the CP/MISF (critical path/most immediate successors first) heuristic described by Kasahara (reference section 2.4.1) is a subset of our dynamic priority heuristic. Our task execution deadline priority is equivalent to Kasahara's critical path priority and our descendant degree priority is equivalent to Kasahara's MISF priority. Our heuristic provides for additional constraints (e.g. communication and memory) as well as nonhomogeneous processors. The key difference which allows these additional constraints to be accommodated by our heuristic

is the dynamic priority computation which continually adjusts to the previously allocated tasks.

The lookahead extension is implemented by adding the almost ready tasks to the set of ready tasks discussed above. An almost ready task must have all precedence relations satisfied except for one or more executing antecedents. These execution antecedents must complete execution during a defined lookahead time window. Thus, an almost ready task is guaranteed to become ready during the time period defined by the lookahead window. If an almost ready task has a sufficiently high priority, then an idle processor will be forced to remain idle (i.e., reserved for the almost ready task) until the almost ready task becomes ready.

If an almost ready task is chosen as the highest priority task on a given processor, that processor is "assigned" the almost ready task which forces the processor to be idle until the next scheduling event (since the almost ready task can't begin execution yet).

The factors which control the lookahead extension are:

- o Lookahead window - period of time used in lookahead computation
- o Lookahead weight - fractional weight to reduce the priority of almost ready tasks in comparison to ready tasks

The dynamic priority scheduler procedure is shown in Figure 16. The procedure first calls INIT SEQUENCE to initialize the event-based simulation. Then INIT PRIOR is called to compute the initial task priorities. INIT PRIOR computes the priorities of all tasks which have no antecedents and are therefore ready to start at the first event.

```
procedure DYNAMIC.PRIORITY (PROBLEM,SCHED)
; PROBLEM      - input definition of application and
;                architecture characteristics/constraints
; ALL          - allocation mapping
; FEAS.SEQ     - boolean denoting feasible sequence
; SEQ          - sequencing of allocation
; SCHED        - complete schedule = (ALL,SEQ)
;subroutines called
; INIT.SEQUENCE          -  initialize sequence variables for the
;                             current allocation
; INIT.PRIOR             -  initialize task priority parameters
; FIND.NEXT.SEQUENCE     -  searches forward to find a new feasible
;                             sequence for the current allocation
; REPORT                 -  reports the optimal feasible schedule
;                             or reports no feasible schedules exist


  call INIT.SEQUENCE
  call INIT.PRIOR
  call FIND.NEXT.SEQUENCE'  (ALL,SEQ,FEAS.SEQ)
  if (FEAS.SEQ) then
     set SCHED = (ALL,SEQ)
     call REPORT (SCHED)
  else call REPORT (false)

end procedure
```

Figure 16. Dynamic Priority Heuristic Procedure.

INIT PRIOR also computes some of the priority factors (those that have a constant value regardless of the sequence of events) for all tasks. The procedure then calls FIND NEXT SEQUENCE, once, to find the task allocation and schedule. As for the constraint relaxing heuristic, FIND NEXT SEQUENCE performs the event-based simulation and is modified by using the NEXT SEQ version shown in Figure 15. For this dynamic priority heuristic, however, the GET HIGH PRIORITY subroutine makes all of the allocation and scheduling decisions based on task priorities, rather than on a relaxed schedule.

3.4.2 Dynamic Priority Subroutines

The dynamic priority procedure requires two new subroutines: INIT PRIOR and GET HIGH PRIORITY. INIT PRIOR determines the initial task priority as described above. The GET HIGH PRIORITY subroutine is shown in Figure 17. It first determines the set of ready tasks, almost ready tasks, and idle processors. It then enters a loop in which either all of the ready tasks are allocated to a processor, or all of the processors have tasks allocated to them. At each iteration of the loop, all of the ready and almost ready tasks are evaluated for all of the idle processors. The highest priority combination of task and processor is determined and that task is assigned to that processor. The task and processor are then eliminated from their respective sets and the loop continues until one of the sets is empty.

```
subroutine GET.HIGH.PRIORITY
;variable definition
;  WINDOW      -  size of lookahead window
;  READY       -  set of tasks currently ready
;  ALMOST      -  set of tasks becoming ready during window
;  IDLE.PROC   -  set of idle processors
;  HIGH.TASK   -  ready or almost ready task with hi priority
;  HIGH.PROC   -  processor on wich HIGH.TASK has priority
;subroutines called
;  FIND.READY.TASKS  -  find set of ready and almost ready tasks
;  FIND.IDLE.PROC    -  find set of idle processors
;  FIND.HIGH.TASK    -  find highest priority ready or almost
;                       ready task for all idle processors
;  ASSIGN.HIGH       -  reserve the HIGH.PROC for the HIGH.TASK
;                      (HIGH.PROC is idled if HIGH.TASK almost ready)
;
  call FIND.READY.TASKS (WINDOW, READY, ALMOST)
  call FIND.IDLE.PROC (IDLE.PROC)

  do while ( (IDLE.PROC not empty) and (READY not empty) )
    call FIND.HIGH.TASK (HIGH.TASK,HIGH.PROC, READY, ALMOST, IDLE)
    call ASSIGN.HIGH (HIGH.TASK, HIGH.PROC)
    set IDLE.PROC = IDLE.PROC - HIGH.PROC
    if (HIGH.TASK member READY) then set READY = READY - HIGH.TASK
    else set ALMOST = ALMOST - HIGH.TASK

return
```

Figure 17. GET HIGH PRIORITY Subroutine for Dynamic Priority.

3.4.3 Dynamic Priority Time and Space Complexity

The time complexity of this heuristic is driven by the subroutine GET HIGH PRIORITY which computes the priority for each ready and almost ready task and idle processor. At any given time, n tasks could be ready and m processors could be idle, requiring O(n*m) calculations to determine the highest task for one processor. This is then repeated for each idle processor, requiring O $(n*m^2)$. The event simulator calls GET HIGH PRIORITY as each task is scheduled, giving a total complexity of $O(n^2*m^2)$. This type of polynomial complexity is acceptable in order to schedule large numbers of tasks and processors in a reasonable amount of computational time.

The space complexity of this heuristic is largely determined by the space required to store the input definition of the problem, $O((n+m)^2)$. The list scheduler simulation maintains the status of each processor using O(m) space and maintains the status of each task using O(n) space. The priority calculation equations use a constant space since the task priorities are computed in sequence and only the highest is saved. For an application of this heuristic to a realtime scheduler, some of the priority components could be precomputed and stored for each task and processor, thus trading off O(n*m) space for reduced computation time.

CHAPTER 4 SCHEDULING ALGORITHM RESULTS AND ANALYSES

The three algorithms discussed in Chapter 3 were coded in FORTRAN 77 and executed on a VAX computer. This chapter discusses the results gathered by exercising these algorithms on a variety of test cases. The results are used to characterize and compare the scheduling performance and time complexity of the different algorithms.

4.1 Empirical Procedure

The results are gathered by using a given scheduling algorithm to schedule a set of scheduling instances. Each scheduling instance specifies all of the task and processor characteristics (execution time, deadlines, communication distances, etc.) needed for the scheduling problem. For each scheduling instance, the total schedule length is recorded if a feasible schedule is found. The number of scheduling nodes examined is also recorded to measure the computation time required for the schedule. A scheduling node is either an allocation or sequence node in the respective search trees.

A large set of instances is required to develop a good measure of the algorithm performance for comparison or prediction purposes. We developed an "instance generator" which randomly creates scheduling instances from user-supplied bounds for each of the problem characteristics: task execution length, amount of communication,

probability of precedence links, and so on. The instance generator randomly assigns specific values with a uniform distribution between the user-supplied upper and lower bounds. Thus the random execution time will fall between the execution bounds and the random communication time will fall between the communication bounds, etc.

The random precedence relationships are created by randomly defining direct precedence links between tasks. The user-supplied precedence percentage defines the probability that a precedence link will be specified between each Ti and Tj for i = 1 ... N-1 and j = i+1 ... N. To keep the tasks in precedence related order, a task Ti can be the antecedent of Tj (i.e., Ti <* Tj) only if i < j. Thus the precedence matrix is always upper triangular and all precedence relationships are acyclic. Once this initial precedence matrix is created, all redundant precedence links are removed so that Ti <* Tj implies that there is no Ti <* Tk and Tk <* Tj for all i. As an example, if T1 <* T2 and T2 <* T5 and T1 <* T5, then T1 <* T5 is redundant and is removed. The final precedence matrix defines those pairs of tasks which have a direct precedence relationship and which may have intertask communication (using the communication bounds to determine the amount of communication).

An example execution time, communication time, and precedence percentage is given by:

Execution time: lower bound = 200, upper bound = 8500

Communication time: lower bound = 500, upper bound = 4000

Precedence: percentage = 60%

Two example scheduling instances created using these controls are given in Figure 18. Note that all execution times are between the bounds (200,8500), the precedence matrix is upper triangular, and the communication values are within the communication bounds (500,4000). The tasks with communication correspond to the precedence matrix since communication occurs only between tasks with direct precedence links.

This generator was set up to produce many random instances for a specific number of tasks and processors. Most of the following results examine the importance of a particular variable for a range of tasks and processors and each sample point represents the performance for a particular number of tasks and processors. For a given sample point, several instances are generated and evaluated using the scheduler. The average of the results is used to characterize that sample point. When comparing two different scheduling algorithms, the exact same set of cases is used for each algorithm by manipulating the random number generator seed value.

4.2 Optimal Scheduler Performance

4.2.1 Optimal Scheduling Example

This section uses the image generator scheduling problem discussed in Chapter 1 to illustrate the operation of the optimal scheduler. We give some of the allocations and sequences which were examined by the scheduler to determine the optimal schedule. The optimal schedule length is shown to be 14500 time units. This is the same length as the

OF TASKS(N) = 8 # OF PROCS(M) = 3

Q MATRIX

TASK	PROCESSOR 1	2	3
1	1948	1488	499
2	5085	1542	5786
3	3848	3451	3478
4	4423	3990	1531
5	5771	3150	960
6	2523	4539	5326
7	3897	5722	8360
8	5338	4448	4349

PRECEDENCE MATRIX FOR N= 8

	1	2	3	4	5	6	7	8
1	0	1	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0
3	0	0	0	1	1	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

COMMUNICATION MATRIX FOR N= 8

	1	2	3	4	5	6	7	8
1	0	3750	0	0	0	0	0	0
2	0	0	3536	0	0	0	0	0
3	0	0	0	2572	1746	0	0	0
4	0	0	0	0	0	1598	0	0
5	0	0	0	0	0	0	0	1901
6	0	0	0	0	0	0	3633	3181
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

DISTANCE MATRIX

	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

OF TASKS(N) = 8 # OF PROCS(M) = 3

Q MATRIX

TASK	PROCESSOR 1	2	3
1	6635	8238	7110
2	1672	4838	6478
3	2223	7539	1130
4	2470	2005	7194
5	249	1447	7241
6	6778	1140	4735
7	5504	4332	2159
8	4242	4385	5532

PRECEDENCE MATRIX FOR N= 8

	1	2	3	4	5	6	7	8
1	0	0	1	1	0	0	0	0
2	0	0	0	1	0	0	0	0
3	0	0	0	0	0	1	0	1
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

COMMUNICATION MATRIX FOR N= 8

	1	2	3	4	5	6	7	8
1	0	0	2085	2572	0	0	0	0
2	0	0	0	2415	0	0	0	0
3	0	0	0	0	0	1603	0	1545
4	0	0	0	0	810	0	0	0
5	0	0	0	0	0	2557	0	836
6	0	0	0	0	0	0	2239	0
7	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

DISTANCE MATRIX

	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

Figure 18. Random Instances Created by Random Instance Generator.

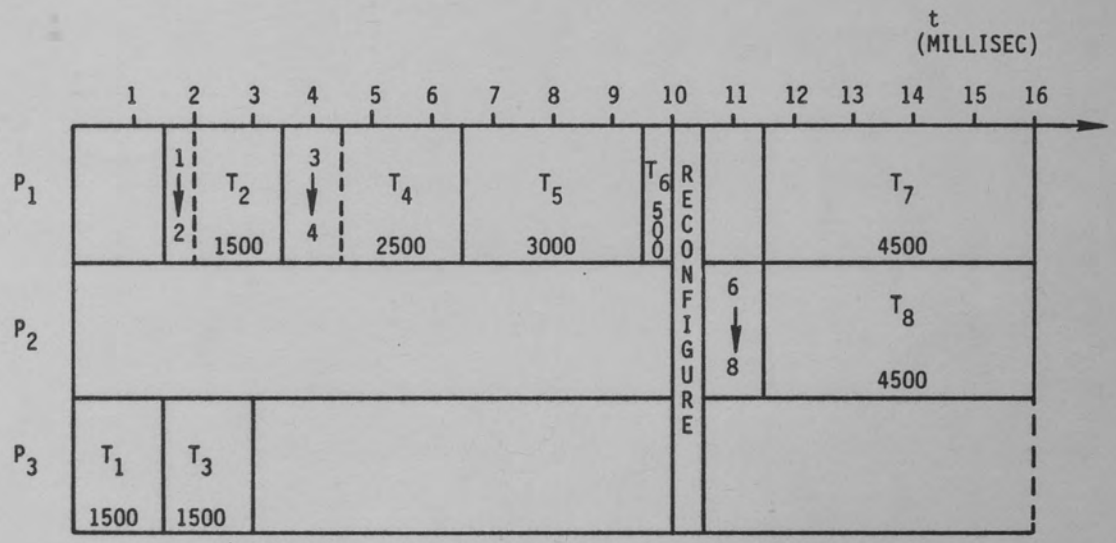
schedule given in Figure 5 of Chapter 1, although the two optimal schedules differ in the allocation of tasks 5, 6, 7, and 8.

The optimal scheduler begins by finding the first feasible allocation. There are a total of 10,935 possible allocations ($3^8*5/3$) for the 3 processors, 8 tasks and 2 configurations for tasks 7 and 8 (actually only 5/3 configurations since, in the pipeline mode, task 7 and 8 must be on different processors). The first allocation is built by placing each task on the processor which gives the shortest execution and communication time. The first allocation can be represented using the notation of 3.1, where an allocation is a mapping for each task to a processor and configuration:

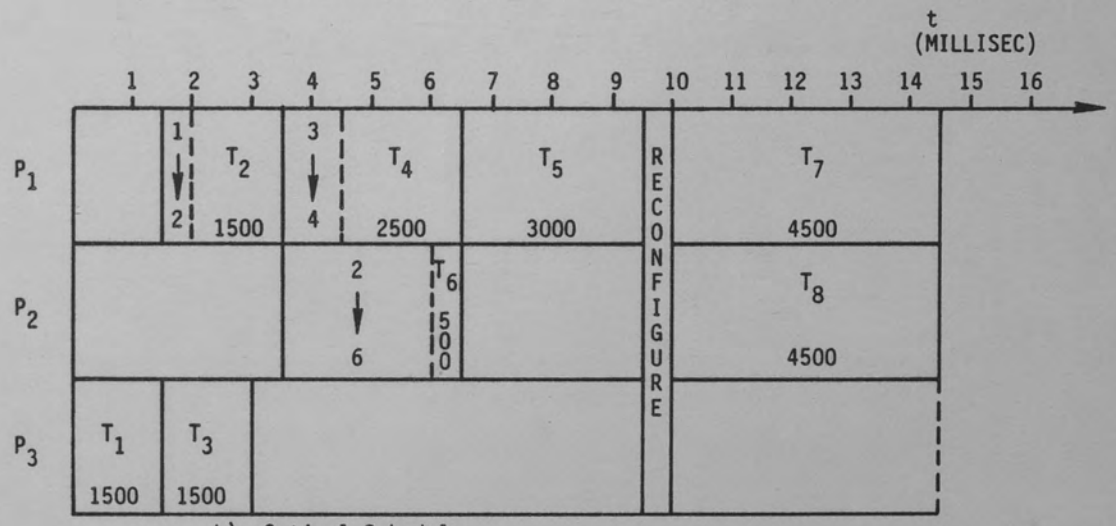
A = ((3,1), (1,1), (3,1), (1,1), (1,1), (1,1), (1,2), (1,2))

This allocation is feasible and a feasible sequence is immediately found which is shown in Figure 19a. The scheduling algorithm records this feasible schedule of 16,000 time units and establishes a new deadline of 15,999 time units. No further sequences of this allocation (e.g., rearranging the execution order of tasks 4, 5, and 6) are feasible since the resulting schedule length is at least 16,000 units.

When all of the sequences of feasible allocation #1 are exhausted, a new feasible allocation is found. New feasible allocations are found by allocating tasks 7 and 8 to different processors, but this does not improve the schedule length. The ninth feasible allocation allocates task 6 to a different processor (P2) which leads to a feasible sequence with length 14,500 as shown in Figure 19b. This feasible schedule is recorded and the deadline is reduced to 14,999.



a) Schedule resulting from first feasible allocation.



b) Optimal Schedule.

Figure 19. Optimal Scheduler Solution of Example Problem.

No other feasible sequences can be found with a schedule length less than 14,500, so this is optimal. A total of 190 feasible allocations were found and tested, but only allocations #1 and #9 led to feasible schedules as shown. In order to find those 190 feasible allocations, a total of 462 allocation nodes were searched in the forward direction. Note that a full allocation tree of 10,935 leaf nodes ($3^8*5/3$) has an additional 5,466 internal nodes ($2* (3^7 + \ldots + 3^1)$). Thus, by pruning the allocation tree using the available constraints, only 462 / 16,401 or 3% of the tree nodes were searched.

For the two feasible sequences, 2,272 sequencing nodes were checked. These sequences included permuting tasks on the same processor (such as tasks 4, 5 and 6 on P1 of allocation #1) and introducing idle times before starting any task (e.g., delaying the start of task 2 until task 3 finishes in case a dependent of task 3 should precede task 2). For allocation #1 there are a total of 174 million sequences for the tasks ((2*5)[(2*1)[(2*2)[) where each task can be preceded by idle time. Most of the sequences are never considered because they violate precedence rules. Our algorithm had to consider only a small fraction (2,272 / 174 million) by enforcing the precedence rules and checking task deadlines as tasks were scheduled.

The total number of nodes our algorithm searches is therefore 2,734 (462 allocation + 2,272 sequence). This is a good measure of the computational time required since the computations required at each node are roughly constant. (There are some search functions dependent on the number of tasks and processors but these functions are not a

significant component overall.) A larger scheduling problem searching 4 million nodes requires about 1/2 hour run time (15 minutes CPU) on the VAX 8600 under VMS. This equates to 450 microsec per node (225 CPU microsec). Note that only forward nodes are counted and every forward node is subsequently backtracked. Thus the time per node includes both the forward and backtrack computations.

4.2.2 Optimal Scheduler Evaluation

The optimal scheduler was exercised for a variety of random cases. This section presents the statistics gathered for over 2000 test problems. These statistics will serve as an optimal baseline against which we compare our priority scheduling algorithm and the versions of the constraint relaxing algorithm corresponding to other researchers' approaches. The comparison is done later in 4.3. This section examines the optimal results themselves to characterize how the general characteristics of sets of random cases affect the average schedule length and average scheduling time (i.e., number of nodes searched to find the schedule).

The statistics shown in figures 20 through 24 record the average schedule length and average scheduling nodes as a function of m, the number of processors (independent axis), n, the number of tasks (family of curves) and p, the precedence percentage (different graphs within a figure). The execution and communication bounds are fixed for any figure. Each graph is a set of sample points linearly connected according to the number of tasks in the sample. The schedule length,

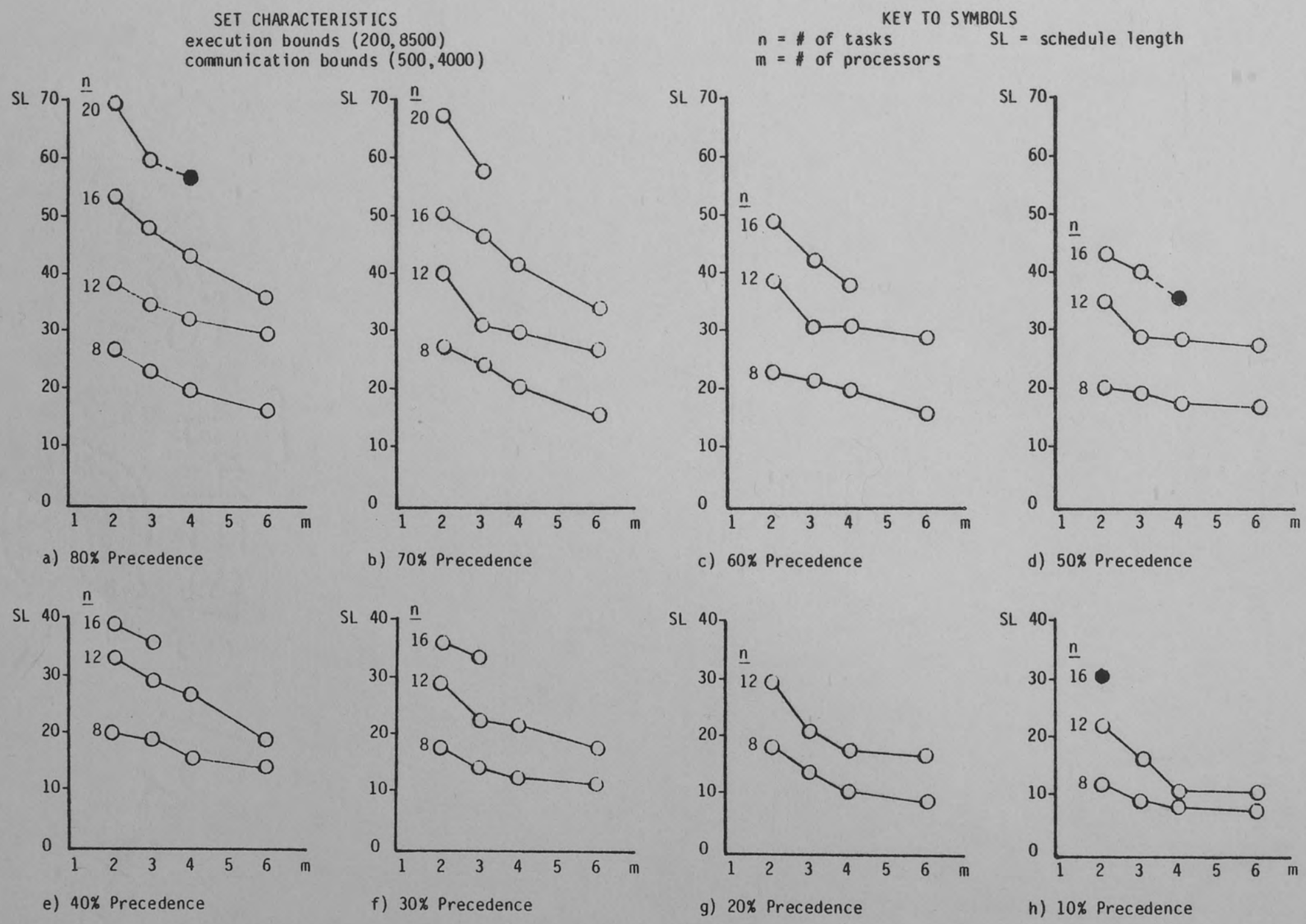


Figure 20. Set 1 Optimal Schedule Length Results.

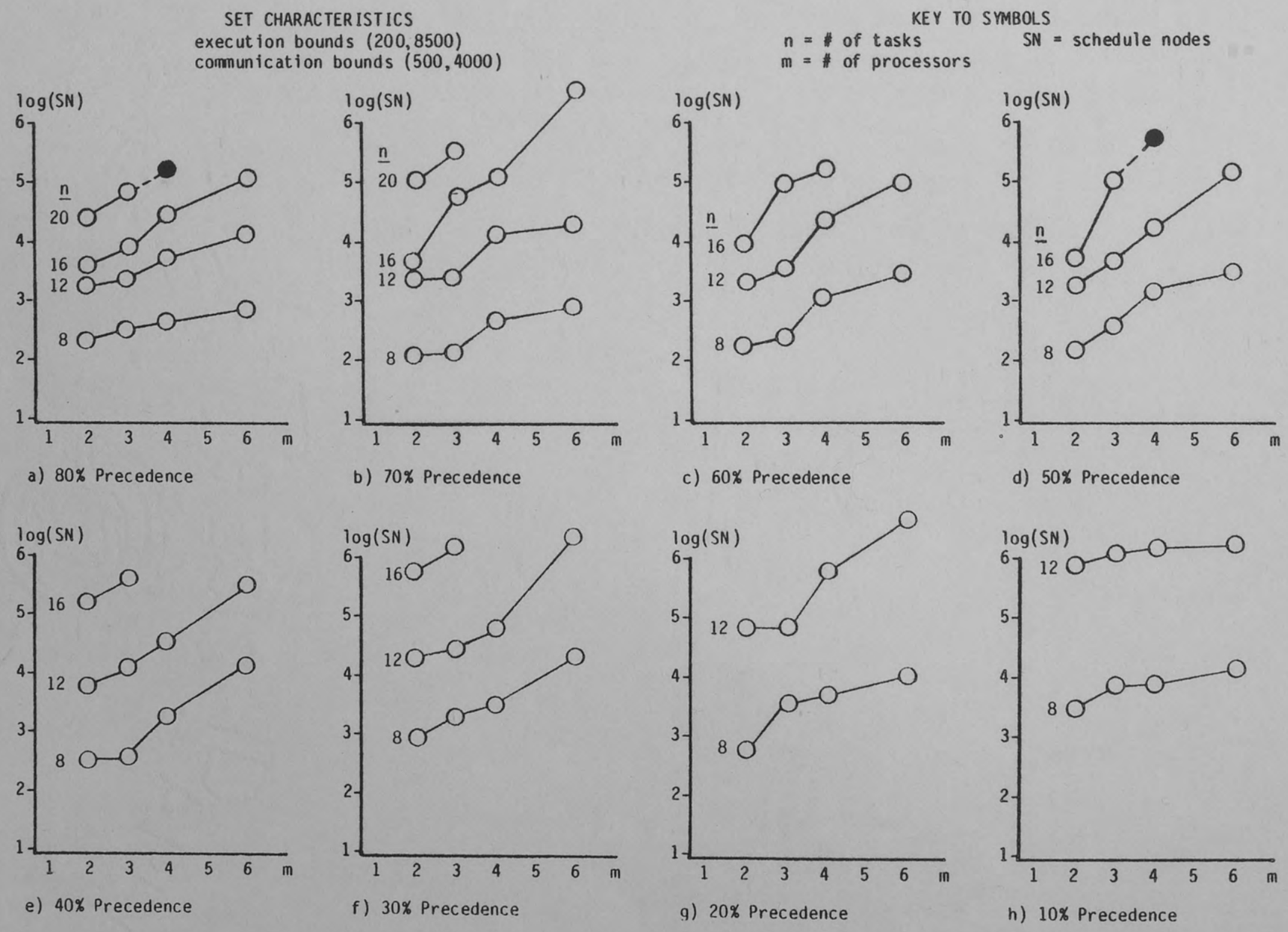
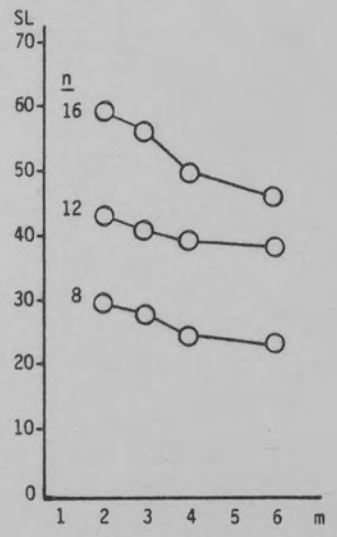


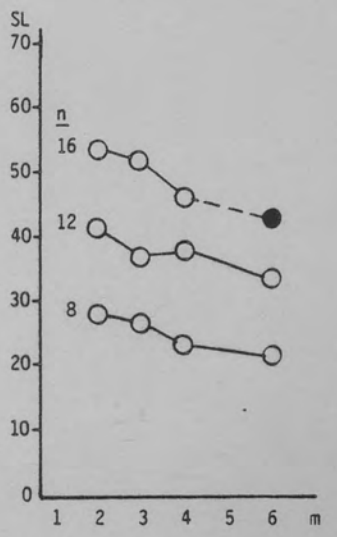
Figure 21. Set 1 Optimal Schedule Node Results.

SET CHARACTERISTICS
execution bounds (200,8500)
communication bounds (2000,5000)

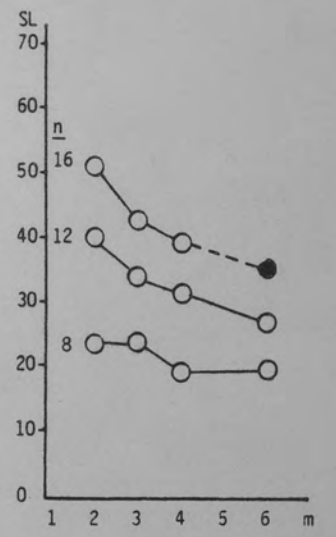
KEY TO SYMBOLS
n = # of tasks
m = # of processors
SL = optimal schedule length
SN = # nodes to compute schedule



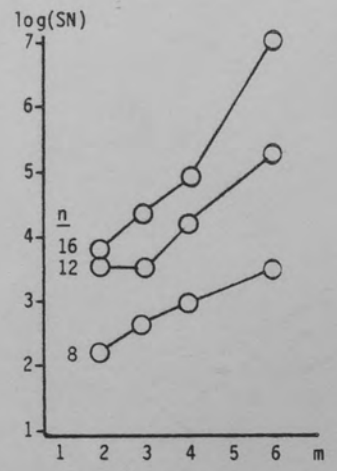
a) Scedule Length for
80% Precedence



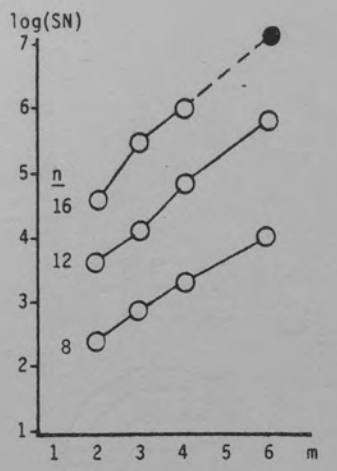
b) Schedule Length for
60% Precedence



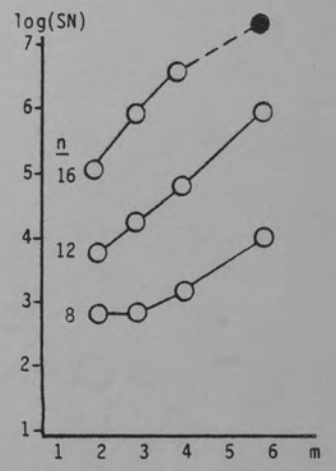
c) Schedule Length for
40% Precedence



d) Schedule Nodes for
80% Precedence



e) Schedule Nodes for
60% Precedence

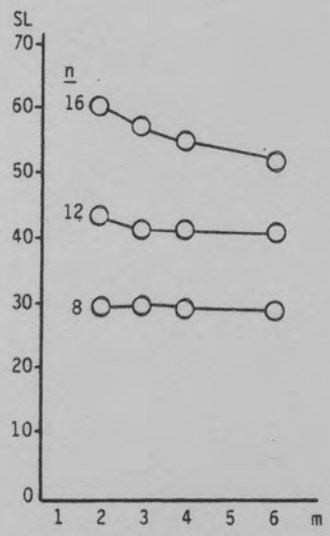


f) Schedule Nodes for
40% Precedence

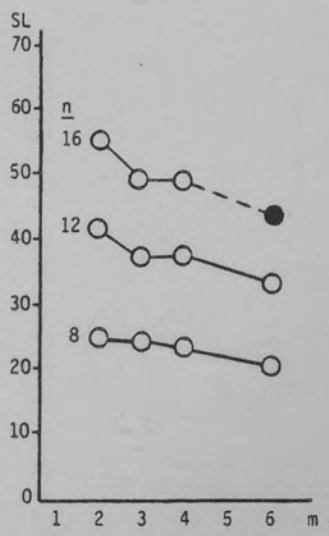
Figure 22. Set 2 Optimal Results - More Communication Time.

SET CHARACTERISTICS
execution bounds (2000,6700)
communication bounds (500,4000)

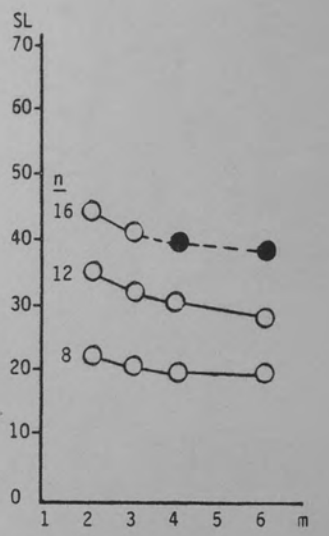
KEY TO SYMBOLS
n = # of tasks
m = # of processors
SL = optimal schedule length
SN = # nodes to compute schedule



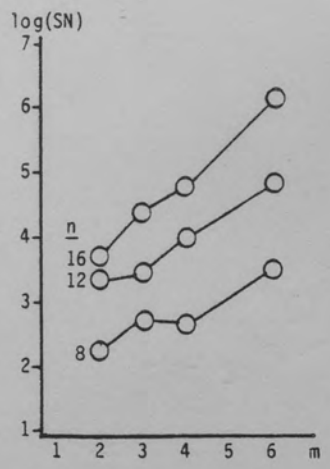
a) Scedule Length for
80% Precedence



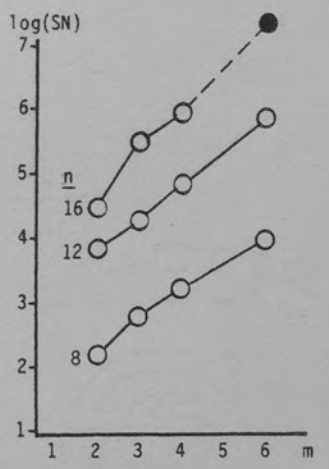
b) Schedule Length for
60% Precedence



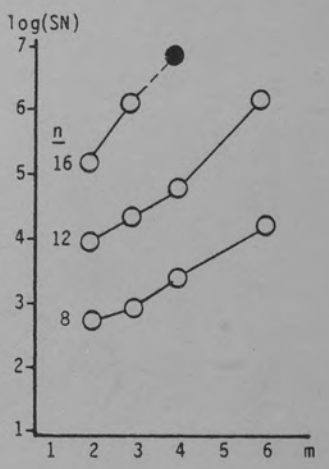
c) Schedule Length for
40% Precedence



d) Schedule Nodes for
80% Precedence



e) Schedule Nodes for
60% Precedence

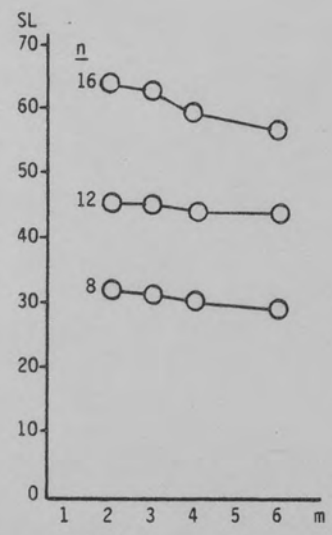


f) Schedule Nodes for
40% Precedence

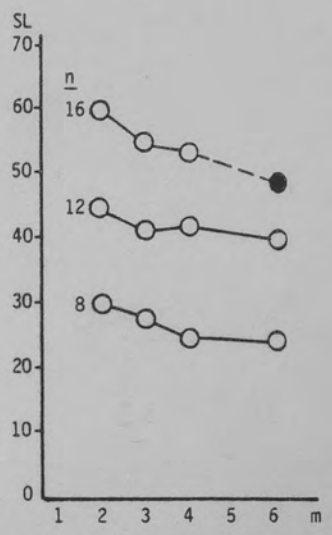
Figure 23. Set 3 Optimal Results - Less Execution Variance.

SET CHARACTERISTICS
execution bounds (2000,6700)
communication bounds (2000,5000)

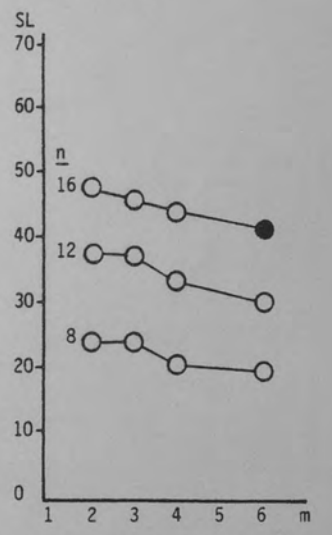
KEY TO SYMBOLS
n = # of tasks
m = # of processors
SL = optimal schedule length
SN = # nodes to compute schedule



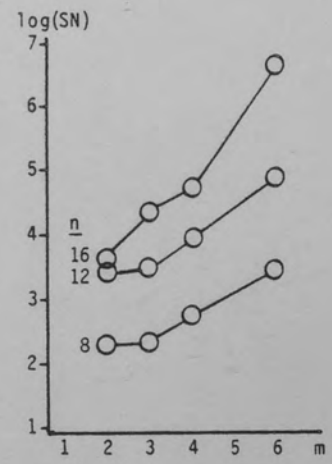
a) Scedule Length for
80% Precedence



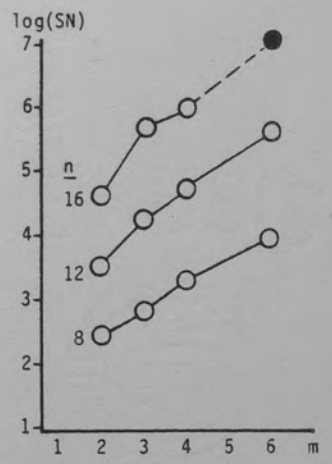
b) Schedule Length for
60% Precedence



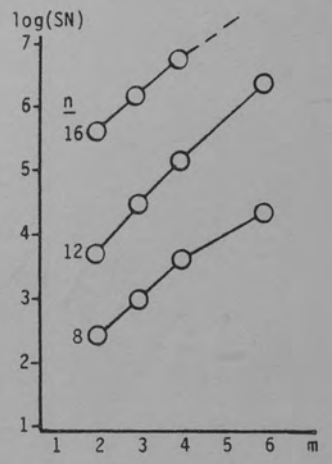
c) Schedule Length for
40% Precedence



d) Schedule Nodes for
80% Precedence



e) Schedule Nodes for
60% Precedence



f) Schedule Nodes for
40% Precedence

Figure 24. Set 4 Optimal Results - More Communication, Less Execution.

SL, is shown in thousands of time units. The number of scheduling nodes is shown on a logarithmic scale because of the exponential character. The graphed value for the number of scheduling nodes, SN, is defined by SN = log 10 (number of nodes). Thus SN = 6 corresponds to 1 million nodes searched.

The sample point is the average value for 10 random cases created using the specified number of tasks, number of processors, precedence percentage, execution bounds, and communication bounds. Some sample points are the average of fewer than ten cases, and this is indicated on the graphs using a dotted line and solid sample point. This condition occurs when the computational time required to find the schedule for all ten cases exceeded our computational limits. The partial results are therefore given as an approximation to the full set of ten cases.

All scheduling cases used nonhomogeneous processors with a simple cross bar type communication network (unit distance between processors and zero distance within a processor). The memory constraints were defined so that 70% of the tasks could be allocated to a single processor. The deadline for all tasks was set equal to the combined average sequential execution time of all tasks.

The first set of results (Set 1) are given in figures 20 and 21. Figure 20a-h shows the average schedule length for scheduling instances with task execution bounds of (200,8500), communication bounds of (500, 4000) and precedence percentages ranging from 80% to 10% for a-h,

respectively. The results are to be expected that more processors and more task concurrency (smaller precedence percentages) lead to shorter average schedule length. Even the cases which are highly precedent constrained (e.g., 80% precedence in Figure 20h) show schedule length improvements with more processors. This is because the processors are nonhomogeneous, so adding processors may result in a particular task executing faster on the added processor. This type of allocation based on minimizing each task's execution time is partially offset by the added communication between processors, but provides a net decrease in the schedule length.

We found the variance in schedule length (within a sample of 10 cases) to be about 10% of the schedule length. This small variance is representative of all the optimal results reported here. A small variance indicates that a fairly good estimate can be developed based on the general application characteristics (execution time variance, communication variance, precedence, etc.) without detailed characteristics of each task. Although all of our scheduling algorithms require the detailed task characteristics to develop the schedule, some applications could benefit from a good estimated schedule length.

The computational time measure for the Set 1 schedules are given in Figure 21a-h. The average number of computational (or schedule) nodes are given for each sample point of ten schedules. For the different degrees of precedence, one can use this information to estimate the largest size problem which can be solved using a specific amount of computer resources. For a given precedence percentage, the

scheduling nodes increase by nearly an order of magnitude when the tasks increase by four. We imposed a computational limit of 4 million nodes because of the large number of cases we processed (i.e., up to 40 million nodes for the ten schedules in one sample point). One can predict, for example, that to schedule twenty tasks on three processors with 30% precedence would require an average of ten million nodes. This is near the practical limit. However, the actual scheduling times varied widely about the average, with the variance frequently exceeding the mean. Thus the hypothesized case with twenty tasks on three processors with 30% precedence might require 100 million nodes or only 500,000.

The Set 2 problem characteristics are identical to Set 1 except the task communication is increased relative to the task execution time. In Set 2 the communication bounds are (2000,5000) so the average communication is 3,500 and the variation in communication is 1:2.5. For Set 1 the average communication was only 2,250 and the variation was greater (1:8). Figure 22 shows the schedule lengths for 80%, 60%, and 40% precedence. The schedule lengths are approximately 10% longer for two processors due to the increase in average communication. Also note that the schedule length does not decrease as quickly as more processors are added. This is because the added communication discourages scheduling tasks on a different processor just to reduce the task execution time. The number of nodes required to schedule Set 2 is given in Figure 22 and is almost the same as the nodes required for Set 1. This indicates that the scheduling computation time is not very

sensitive to different degrees of communication variation (1:2.5 versus 1:8).

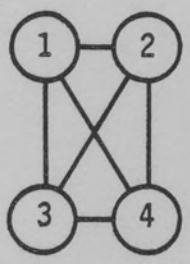
The third set of results characterize a smaller execution time variation (2000,6700) but the same communication variation (500,4000) as Set 1. The average execution is the same but Set 3 has a 1:3.3 variation instead of the 1:42 variation of Set 1. The Set 3 schedule length results in Figure 23 show that the length for 2 processors are about the same as for Set 1, but the schedule length does not decrease rapidly with more processors. This is caused by the smaller variation in task execution lengths which has an equalizing effect on the processors. The number of scheduling nodes for Set 3 are nearly the same as for sets 1 and 2.

The last set of optimal results uses the larger average communication of Set 2, communication bounds of (2000,5000), and the smaller execution variation of Set 3, execution bounds of (2000,6700). The results shown in Figure 24 confirms the earlier observations. The schedule length does not reduce as quickly when the communication increases and task execution variance decreases. The number of scheduling nodes recorded in Figure 24 is approximately the same for all sets and is thus relatively insensitive to changes in task execution and communication time on average.

4.2.3 Optimal Scheduler Application as a Design Tool

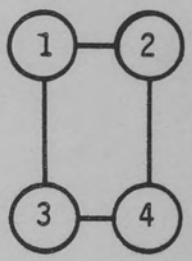
One of the uses of an optimal scheduler is to evaluate how well specific classes of applications will execute on different multiprocessor architectures. This section illustrates this technique by comparing four multiprocessor architectures: crossbar, ring, tree, and star. We determine the average schedule length on each architecture as a measure of their relative ability to support intertask communication. For our test cases we used sixteen tasks and four homogeneous processors. The execution bounds are (200,8500), the communication bounds are (2000,5000), and the precedence values are 60% and 40%. The memory constraint was set to force a distribution of tasks onto all processors. A maximum of 1/3 the tasks could reside on any single processor.

These four communication architectures or configurations are shown in Figure 25a-d for four processors. Next to each configuration is the interprocessor communication (IPC) matrix which is referred to as the distance matrix, D(k,l,r) in Chapter 3. D(k,l,r) defines the communication distance (in time units per word) from Pk to Pl using configuration r. The values of D are computed using the "distance weight" of each communication link between Pk and Pl and the delay added by intervening processors. The distance weight of the links are defined to keep the hardware complexity comparable in all architectures. Thus, the crossbar network with twice as many links has slower links (distance = 2) than the others (distance = 1). The delay added by an intervening processor was defined to reflect the nature of



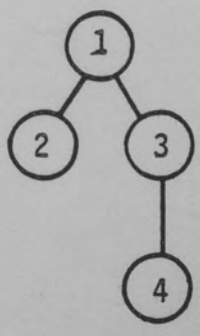
	1	2	3	4
1	0	2	2	2
2	2	0	2	2
3	2	2	0	2
4	2	2	2	0

a) Crossbar and IPC Matrix



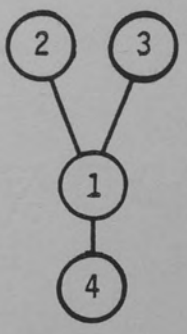
	1	2	3	4
1	0	1	1	4
2	1	0	4	1
3	1	4	0	1
4	4	1	1	0

b) Ring and IPC Matrix



	1	2	3	4
1	0	1	1	2
2	1	0	2	5
3	1	2	0	1
4	2	5	1	0

c) Tree and IPC Matrix



	1	2	3	4
1	0	1	1	1
2	1	0	3	3
3	1	3	0	3
4	1	3	3	0

d) Star and IPC Matrix

Figure 25. Four Communication Configurations.

the specific architecture and varies as discussed below. The distance matrix is always symmetric and the diagonal is zero since communication between tasks on the same processor is assumed instantaneous (e.g., shared memory).

For the crossbar architecture, each processor has a direct link to all others so each distance between processors is 2. For the other architectures, the distance between processors directly connected is 1 and the distance between other processors is the sum of links and delay from intervening processors. For the ring network which generally consists of independent processors, each intervening processor introduces two units of delay, e.g., D(1,3,ring) = 1. The tree architecture is typically designed to efficiently spawn tasks to and retrieve results from immediate descendants. Therefore we defined zero units delay for an intervening processor directly connecting the source and destination processors. If the source and destination are not immediate, then each intervening processor adds one unit of delay. Thus D(2,4,tree) = 5 because of the delay of three links and two intervening processors. The last architecture, the star, uses one unit of delay when passing through the center processor, so the distance is 3 between any two outside processors. Note that the average communication distance is the same for all configurations (24/16 = 1.5). We verified this empirically by randomly scheduling the sixteen tasks onto the processors of the different configurations. When tasks are randomly placed on the processors, all four architectures yield equivalent average schedule lengths.

Figure 26 shows the comparison of the average optimal schedule lengths for the four different architectures. Results were gathered by optimally scheduling a set of 10 cases on each of the four architectures and on a fifth "baseline" architecture, which is our standard crossbar with unit distance between processors (average communication distance of 0.75). Clearly the schedule lengths from each of the four architectures will be at least as long as the baseline. The results for each of the four architectures is represented as a percentage longer than the baseline schedule length to simplify the comparison. These results show that the tree and ring offer average schedule lengths nearly as good as the baseline, even though the average communication distance is twice the baseline. This means that the optimal scheduler is able to schedule tasks so that most interprocessor communication uses the direct communication links with a distance of 1. The star also performs well, but there is some degradation because the fast local links exist only for the center processor. The crossbar with distance weight of 2 performs very poorly, 20% to 35% longer than the baseline.

These results show that, although all four architectures are equivalent for a random scheduling of tasks, a good scheduler can exploit local communication links. A given amount of hardware complexity is better utilized to provide fast local communication links (such as for a tree or ring) even though some paths between processors are quite long (e.g., distances of 4 and 5). This type of local communication is better than guaranteeing a more average performance

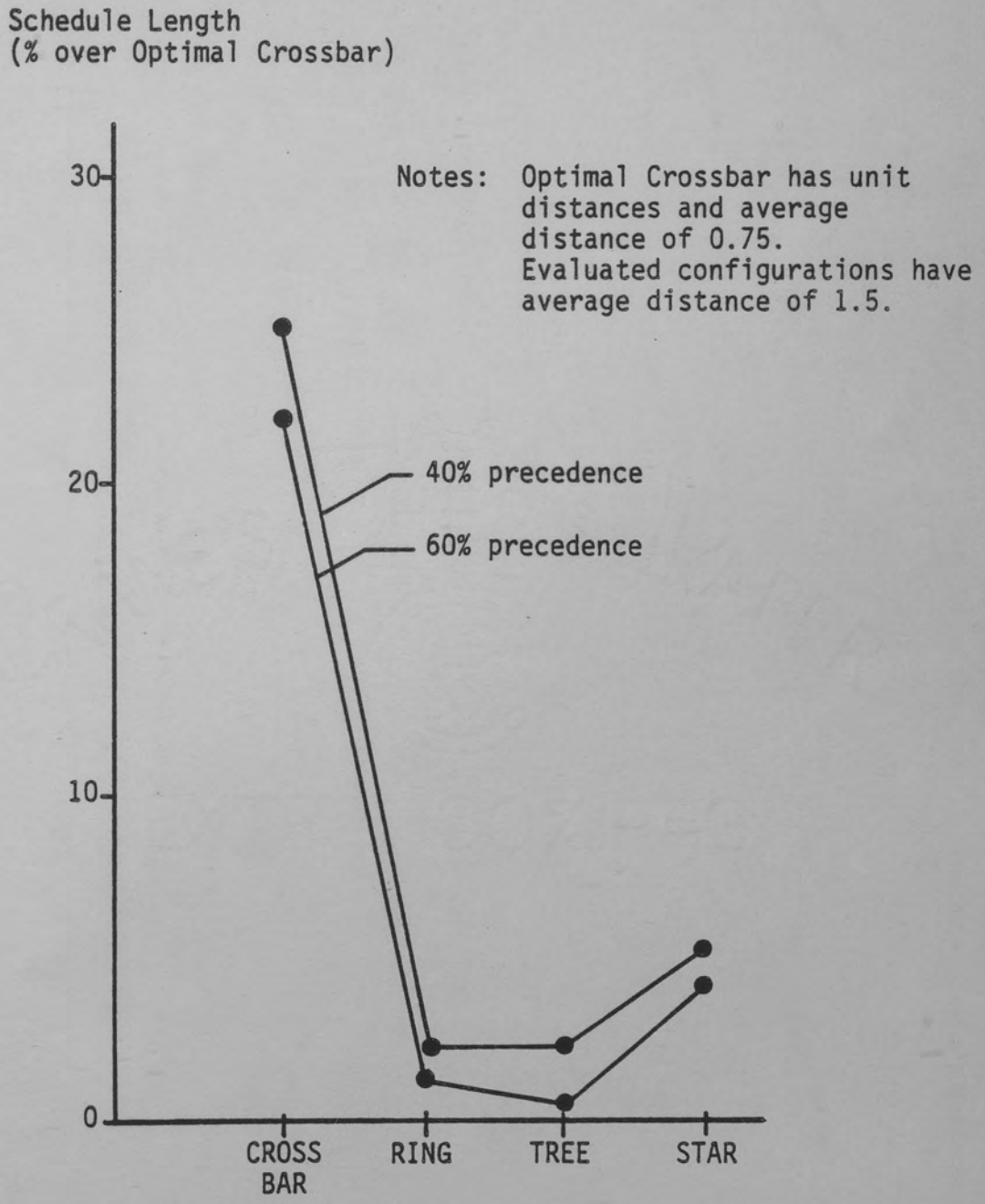


Figure 26. Communication Configuration Schedule Length Comparison.

such as in the star or crossbar. At the same time, even if the tasks are randomly scheduled, the tree and ring will perform at least as well as the others.

4.3 Comparison of Heuristics

This section examines how the constraint relaxing algorithm and priority algorithm compare to each other and to the optimal algorithm. These algorithms were run on a subset of the cases reported in 4.2.2. The exact same set of scheduling problems is used when comparing the performance at a given sample point. Therefore, the nonoptimal algorithms will always produce average schedule lengths (and individual schedule lengths) which are at least as long as the optimal schedule. The average schedule lengths of the nonoptimal algorithms are reported using the percentage over average optimal schedule length.

Three versions of the constraint relaxing heuristic are evaluated, as discussed in 3.3. These versions are denoted COMM, PREC, and EXEC in the following discussion. The COMM version does not consider intertask communication when developing the relaxed schedule and represents the expected results of Kartashev's scheduling approach (reference 2.4.3). The PREC version does not consider task precedence when developing the relaxed schedule, but the communication time which should occur between actual precedence-related tasks is considered. Therefore PREC will tend to cluster tasks with large communication requirements on the same processor. The PREC results represent the expected performance of the graph theory technique (reference 2.2) and integer programming

techniques (reference 2.3). Note that PREC minimizes the schedule length of the relaxed schedule (i.e., maximum sum of execution and communication on individual processors) rather than minimizing the overall sum of execution and communication times on all processors. The third version of the constraint relaxing heuristic, called EXEC, does not consider varying task execution time when developing the relaxed schedule (a constant value is used). This version may be considered for systems with nearly fixed length tasks, but does not directly correspond to an approach suggested by the reviewed researchers.

Two versions of the dynamic priority algorithm were also evaluated. The results labeled PRIOR represent the priority algorithm performance without the lookahead extension. The same set of priority weights was used for all PRIOR results reported here. The weight values used are given by (see 3.4 for definition of weighting functions):

1) task execution weight	- 4
2) processor execution weight	- 40
3) precedence weight	- 32
4) descendence weight	- 4
5) communication weight	- 32
6) deadline weight	- 16
7) memory weight	- 64

While the priority weights could be adjusted to optimize the performance for each schedule, a more realistic approach is to use a standard set of weights for all schedules or perhaps to select a set of weights based on the general characteristics (e.g., execution variance, ratio of communication to execution, etc.). In fact, we generally found that the above weights gave good results for all the cases we attempted

and that varying the weights did not provide significantly better results. The apparent explanation why a single weight set works well is that the weights are applied to the problem specific characteristics (e.g., ratio of a task's execution to the average task execution). Therefore the unique characteristics of the problem are accounted for even though the weights remain the same.

The second version of the priority algorithm we evaluated is the lookahead extension. The results of the lookahead extension are not shown because the extension did not offer a significant improvement over the PRIOR results. This disappointing result is discussed later.

The results of the algorithms are shown in figures 27 to 30. Each of the figures corresponds to the optimal results of one of the four sets discussed in 4.2.2. The measure for schedule length is the percent longer than optimal schedule length, as discussed earlier. For these figures, the important result is the comparison of the different algorithms. Therefore, each curve represents the performance of one particular algorithm for a given number of processors (independent axis) and other problem characteristics fixed for the graph (number of tasks, precedence, execution bounds, etc.).

Figure 27 corresponds to the Set 1 optimal results for 60% and 40% respectively. The Set 1 characteristics are a large variation in execution bounds (200,8500) and a fairly small amount of communication (500,4000). The results show that all of the algorithms degrade as the number of processors increase. While the optimal algorithm was

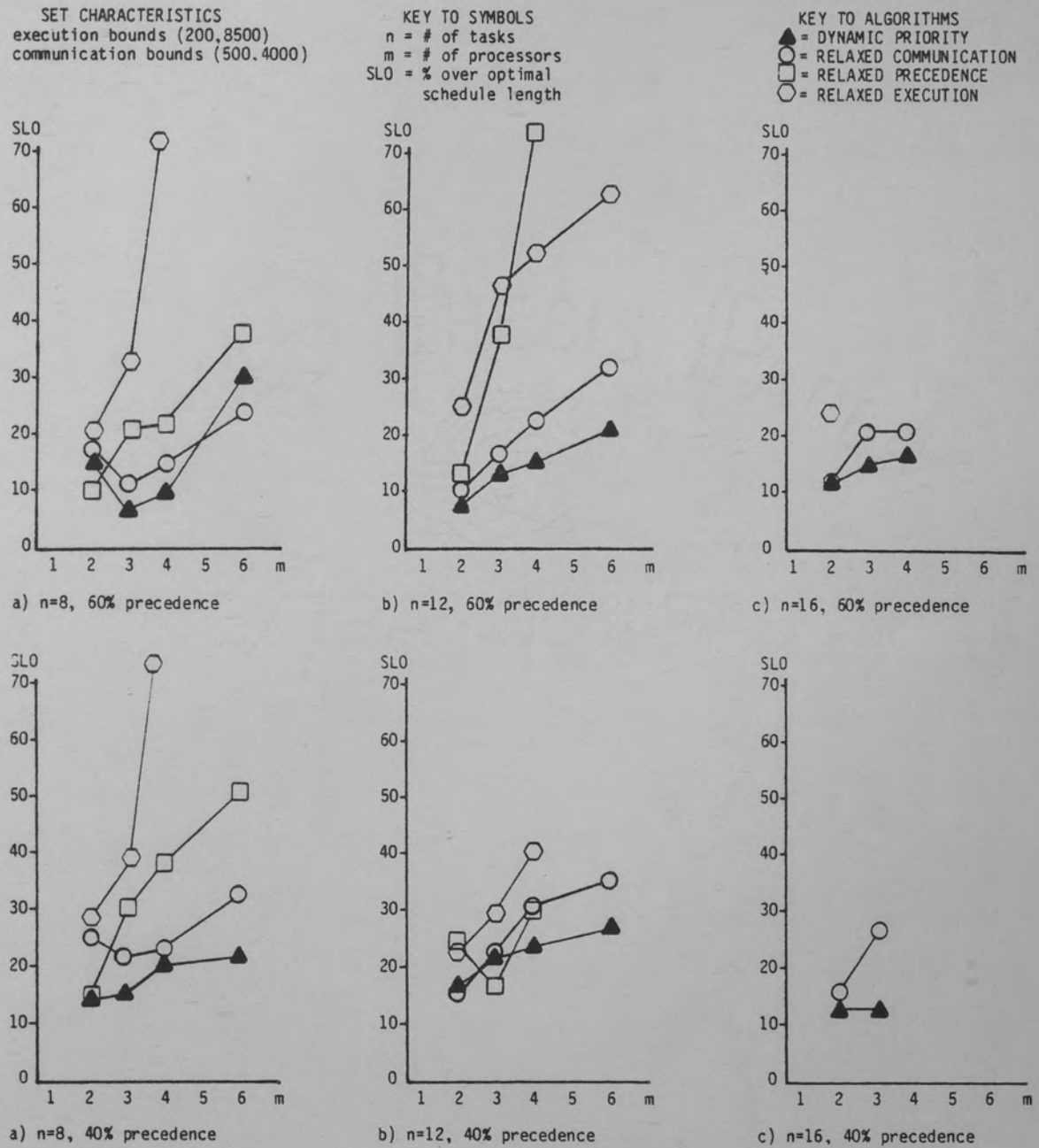


Figure 27. Set 1 Heuristic Schedule Length Results.

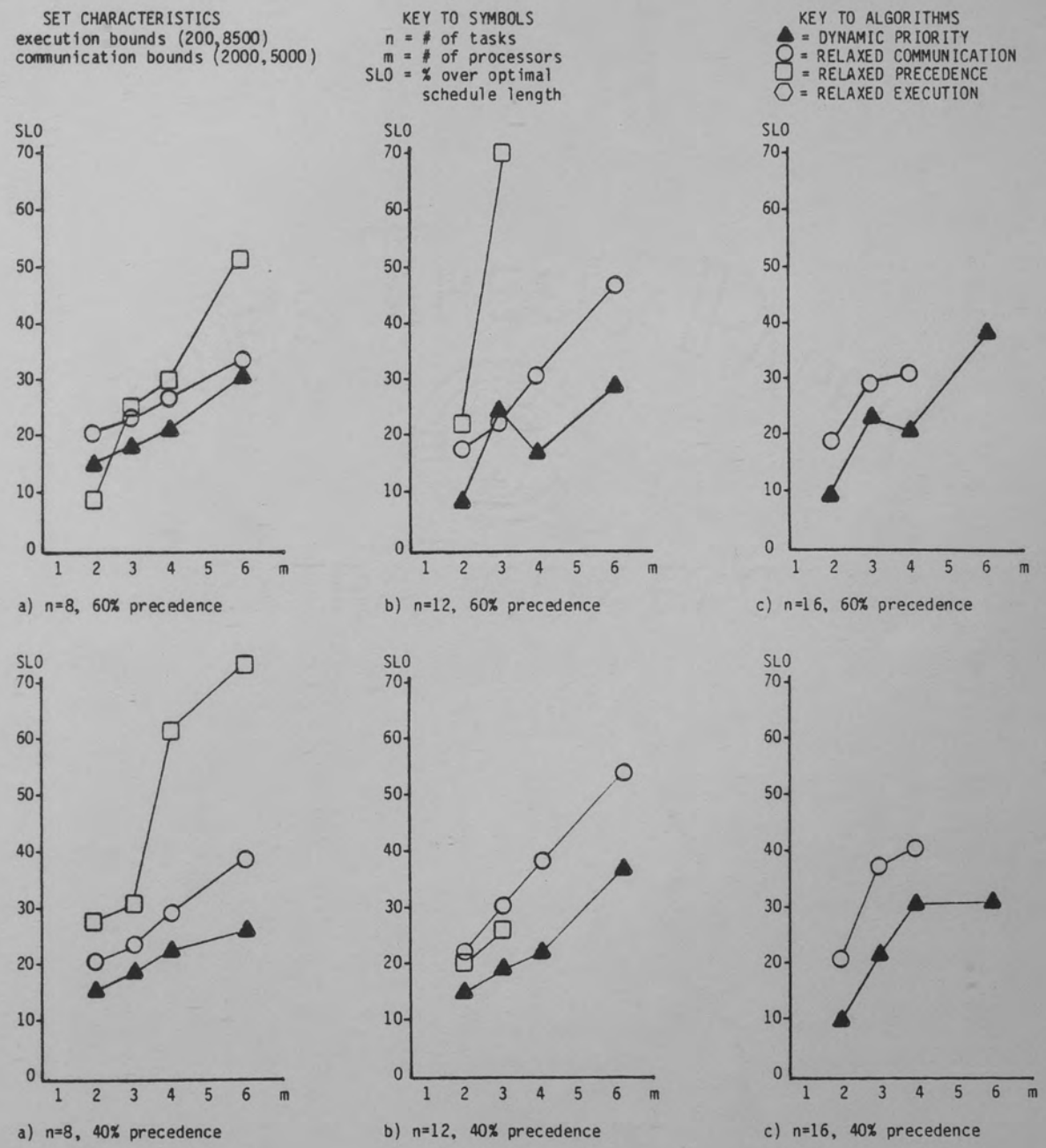


Figure 28. Set 2 Heuristic Schedule Length Results.

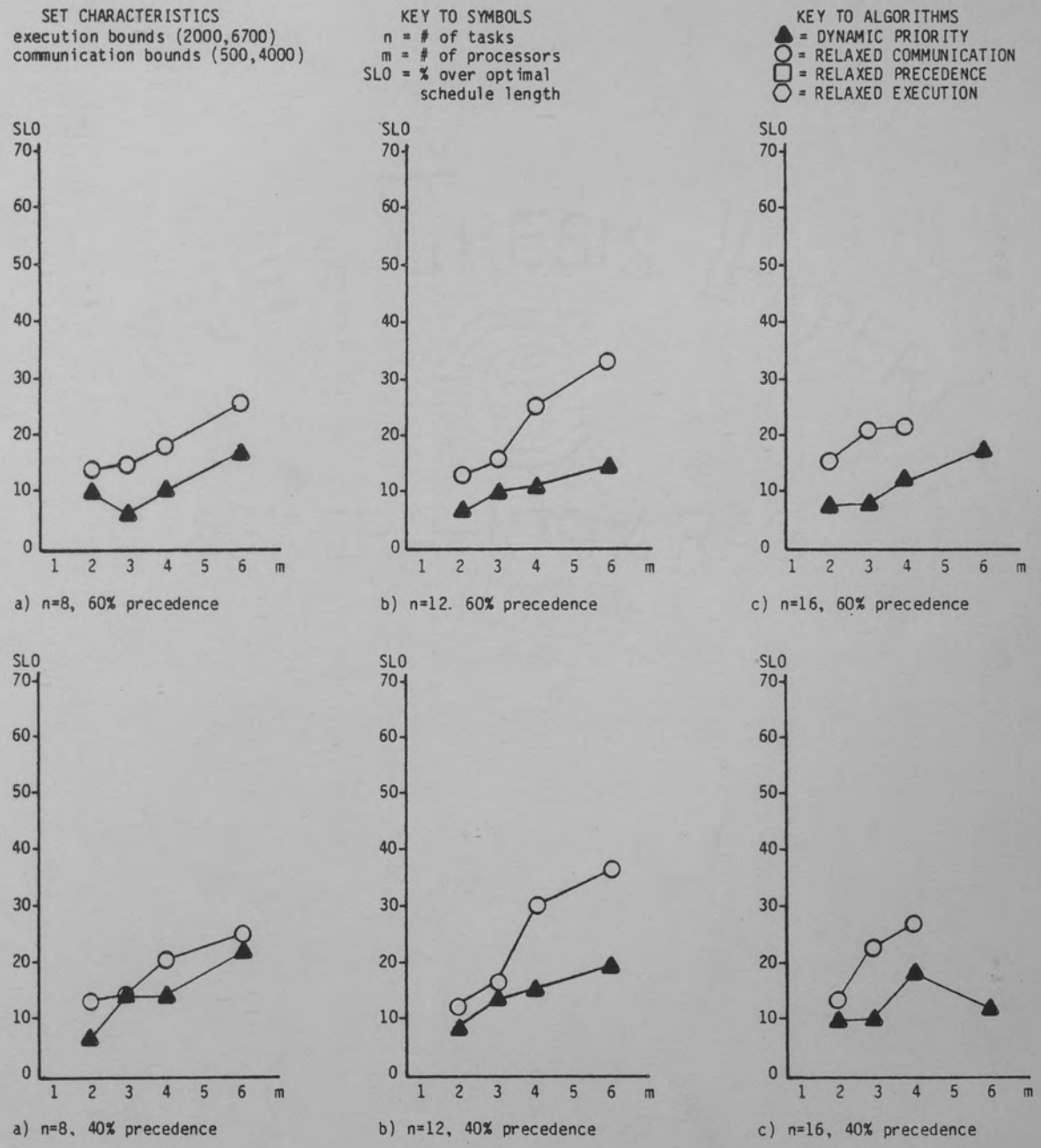


Figure 29. Set 3 Heuristic Schedule Length Results.

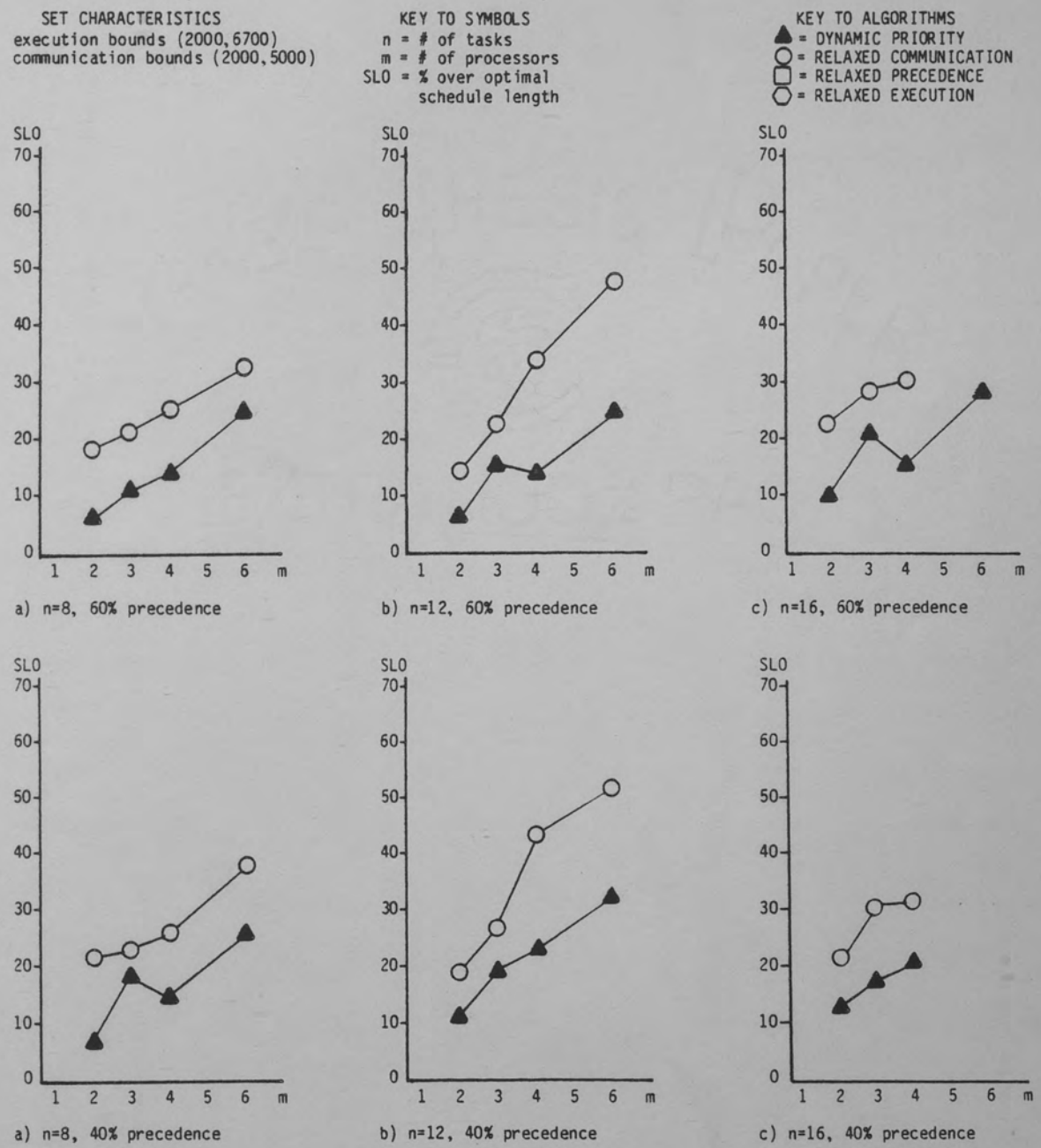


Figure 30. Set 4 Heuristic Schedule Length Results.

consistently able to reduce the schedule length with more processors, these algorithms are not as successful so the percentage over optimal increases. The priority algorithm produces the best schedules, in the range of 10% to 30% over optimal. Note that the performance of PRIOR is nearly the same for the 60% and 40% precedence cases. The next best algorithm is COMM, but COMM degrades noticeably as the precedence decreases from 60% to 40%. As the precedence percentage decreased, the possible concurrency increases and COMM does not perform well with a lot of concurrency. The performance of COMM degrades because, not considering communication, it tends to spread tasks over many processors which increases the communication time. The EXEC and PREC versions of the constraint relaxing algorithm fare the worst and degrade very rapidly as the number of processors increase.

Figure 28 corresponds to the Set 2 optimal results. Set 2 has a larger amount of communication (2000,5000). The PRIOR algorithm continues to perform the best with performance slightly poorer than for Set 1. The COMM algorithm is again second with similar performance to Set 1. The EXEC and PREC algorithms continue to perform very poorly. The same performance trends are shown for the Set 3 and Set 4 results given in figures 29 and 30 respectively.

In summary, the dynamic priority algorithm (PRIOR) performs the best relative to the constraint relaxing versions. PRIOR's absolute performance is in the range of 10% to 40% over optimal schedule length. The performance of PRIOR does degrade as the number of processors increase, as does the performance of all the other nonoptimal

algorithms. (As the number of processors increase, the optimal schedules tend to decrease much faster than the nonoptimal schedules.) The average performance of the priority algorithm is fairly constant over a variety of the other scheduling problem characteristics such as the number of tasks, amounts of communication and execution, and precedence percentage. Again note that the same set of priority weights were used for all results shown.

The priority algorithm lookahead extension did not offer a significant average performance increase. The average performance decreased markedly as the lookahead became larger than approximately one half the average task execution length. For window sizes smaller than this, the lookahead extension had a small impact on the average schedule length, in the range of +/- 2% (measuring the difference between the lookahead percent over optimal and the PRIOR percent over optimal). We examined specific scheduling problems and their solutions to determine the reason the lookahead extension did not improve scheduling performance. The reason is that there were few situations in which the schedule length could be reduced by changing the sequence of an 'almost ready' high priority task with a ready low priority task. This remained true even when there was a large difference in the task priorities. Therefore, the decision to delay a low priority ready task was often wrong or had no effect.

The second best algorithm was the communication constraint relaxing algorithm, representative of Kartashev's approach. This algorithm's performance was generally in the range of 5% to 10% longer

than the PRIOR schedules. (Percentage based on optimal schedule length.) The performance of the COMM algorithm naturally tends to degrade as the communication component becomes more significant, either by reducing the precedence percentage (increasing concurrency) or increasing the requirement for intertask communication. The EXEC and PREC versions of the constraint relaxing algorithm fared the worst for all cases by a wide margin. Obviously these algorithms are not well suited for applications which have those practical constraints.

CHAPTER 5 SUMMARY AND CONCLUSIONS

This chapter summarizes the research approach of this dissertation and briefly reviews our scheduling problem formulation and scheduler algorithm definitions. We draw some conclusions, from the results shown in Chapter 4, concerning the general applicability of the different scheduling algorithms and their relative merits. Finally, we make some recommendations for future research in the multiprocessor scheduling area.

5.1 Dissertation Summary

This dissertation considers the problem of practical constraints in noninterruptible multiprocessor scheduling. The types of constraints generally seen in practical applications and architectures are introduced in Chapter 1, using the image generator example, and a set of scheduling constraints is defined. The related work by previous researchers is reviewed and it is shown that previous researchers address only subsets of our scheduling problem. The previous researchers which did consider many of our constraints used ad hoc scheduling procedures which are not evaluated analytically or empirically.

Our work is a systematic investigation of the scheduling problem and includes the development of an optimal scheduler and the dynamic

priority scheduling heuristic. The optimal scheduler is limited by the exponential computational time complexity. We use it to establish an optimal baseline to measure other scheduling algorithms. Our dynamic priority heuristic achieves good average performance over the measured range of problem characteristics by considering the key scheduling constraints. The dynamic priority heuristic outperforms other scheduling algorithms which do not consider certain key constraints (characteristic of previous researchers' approaches).

Our work formulates the multiprocessor scheduling problem as an allocation and sequencing problem, where an allocation of tasks onto processors is found and then the task execution sequence for that allocation is found. This form is useful for developing an optimal scheduler which uses a double branch and bound technique. The first branch and bound finds all feasible allocations. A feasible allocation is defined to include any allocation leading to a feasible schedule, while excluding most of those allocations which cannot lead to a feasible schedule. Given a feasible allocation, the second branch and bound checks all possible sequences of the tasks on the processors. The sequences are built using an event-based simulation which enforces the precedence constraints, task execution time, communication time, etc. If a feasible sequence of tasks is found (i.e. meets all deadline constraints), then the combination of the feasible sequence and feasible allocation is a feasible schedule. The scheduling algorithm is designed to limit the remaining search to schedules which have a shorter schedule length. The search ends by reporting the optimal

schedule (shortest schedule length) or by reporting that no feasible schedule is possible for the given problem constraints.

We also develop a constraint relaxing scheduling algorithm which allows us to characterize the performance of previous researchers' scheduling approaches. This algorithm can be controlled so that one or more of the scheduling constraints are ignored when developing an initial schedule, called a relaxed schedule. The task allocation and sequence of the relaxed schedule is then used to solve the actual scheduling problem, by reintroducing the constraints, and develop the final schedule. This constraint relaxing algorithm is a valid characterization of other researchers' approaches since it produces an optimal relaxed schedule for the constraints their work considered. By then measuring the performance of the relaxed schedule for the actual problem, we quantize how well the approach works in practical scheduling environments.

The dynamic priority algorithm is then developed. This simple algorithm develops a schedule using an event-based simulation of a list scheduler. The priority of each task is based on several task characteristics, each weighted according to a separate priority weight and the final priority being the summation of the priority components. The priority of the ready tasks at a given point in time is dynamically computed by using the current state of the schedule. A lookahead extension is also described which effectively reserves a processor for a high priority task at the expense of delaying or reallocating a lower priority task.

These algorithms are then evaluated in the results of Chapter 4. The conclusions from these results and the recommendations about future research in this area are given below.

5.2 Applicability of Optimal and Heuristic Schedulers

The purpose of the optimal results reported in 4.2.2 is to establish an optimal baseline against which we compare the heuristics. The general nature of the schedule length results and schedule node results is to be expected. The schedule node results are useful for showing the range of scheduling problem size (number of tasks and processors) which can be optimally solved in a reasonable amount of computational time. A general guideline is that our optimal algorithm can solve problems up to sixteen tasks and four processors in a few hours. We expect that increasing to twenty tasks would increase the computational time by a factor of ten. Therefore, the optimal algorithm could be applicable for non-realtime analysis of some current multiprocessor architectures with four or fewer processors.

A surprising characteristic of the schedule node statistics is that a fairly constant computational time is required for all sets, even though the variations for execution and communication change drastically. Although the worst-case time performance is dependent only on the number of processors, tasks, and configurations, in an actual problem the performance of the branch and bound is greatly affected by the ability to efficiently prune the search trees. We would expect that varying the execution and communication bounds would result in

significantly different computational time requirements. In fact, if any of the bounds are taken to an extreme (e.g., constant execution time, zero communication time) the schedule nodes do increase by about a factor of ten. However, over a normal range of these constraints there seems to be little variation.

An interesting statistic about schedule lengths is the fairly small variance about the average for each group of problems at a given sample point. This small variance suggests that we could extrapolate the observed schedule lengths of solved schedule problems as an estimated schedule length of problems with similar characteristics. Such an estimated schedule length has applications for allocating resources to execute an application, designing systems which can efficiently process certain classes of applications, and developing initial bounds for an actual scheduler. On the other hand, one cannot accurately predict the computation time required to schedule a problem because of the large schedule node variance. Therefore, it is best to anticipate at least a factor of ten variation in the computation time required to solve very similar scheduling problems.

We also demonstrate how the optimal algorithm can be used in architecture design. We measure the performance of different communication architectures and show that hardware resources should be allocated to local communication. The ring and tree architectures have the best performance because they provide fast local communication between different pairs of processors. The importance of a good scheduler is also shown because a random scheduling eliminated the

advantage of the ring and tree. This application of the optimal scheduler to architecture analysis is very exciting because it provides a technique to measure the architecture performance over a wide variety of scheduling problems.

Heuristics are applicable as realtime schedulers and are required for analysis of larger scheduling problems (e.g., thirty-two tasks on four processors). Our results in 4.3 show that our dynamic priority algorithm, based on a simple list scheduling technique, performs well for a variety of scheduling problem characteristics. The performance is especially good for two to four processors. On the other hand, scheduling approaches which do not systematically consider the practical constraints do not perform as well. The approaches which do not consider precedence (such as the integer programming approach) have much poorer performance, even though the time complexity of such a scheduler is much greater than our heuristic. The scheduling approach which does not consider communication performs nearly as well as our heuristic, but the performance decreases as communication becomes more important. These results provide evidence that the scheduler can perform much better when it considers the scheduling constraints, rather than developing a schedule with fewer constraints and attempting to later add in the effect of the constraints.

5.3 Considerations for Future Research

This research can be readily extended in two areas. The first is to use the scheduling algorithms we have developed to evaluate the type

of multiprocessor architecture which is best suited to a particular class of applications. This is a continuation of the work we described in section 4.2.3 where high speed local communication links are shown to be nearly as effective as the more complex high speed global communication links. The potential benefit of further work in this area is a better definition of the types of multiprocessor architectures which will perform well for a variety of cases. This could lead to an approach for automatically configuring a communication architecture to execute a particular application.

The second area for future research is to examine the performance characteristics of our own dynamic priority heuristic and develop techniques to improve the performance. Although our heuristic establishes a measured baseline, it could be improved to produce schedules which are closer to optimal. This improvement could be targeted to a particular set of applications with specific characteristics, or our own approach of developing a generally applicable scheduler could be enhanced. This enhancement process would be a worthwhile activity before applying the heuristic scheduler to an actual application of multiprocessor scheduling with practical constraints.

LIST OF REFERENCES

Bruno, John; Jones, John W.; and So, Kimming. "Deterministic Scheduling with Pipelined Processors." IEEE Transactions on Computers, April 1980, pp. 308-316.

Buehrer, Richard E.; Brundiers, Hans-Joerg; Benz, Hans; Bron, Bernard; Fries, Hansmartin; Haelg, Walter; Halin, Hans Juergen; Isacson, Anders; and Tadian, Milian. "The ETH-Multiprocessor Empress: A Dynamically Configurable MIMD System." IEEE Transactions on Computers, November 1982, pp. 1035-1044.

Chen, Peter Pin-Shan, and Akoka, Jacob. "Optimal Design of Distributed Information Systems." IEEE Transactions on Computers, December 1980, pp. 1068-1080.

Chiang, Y., and Fu, K. "Matching Parallel Algorithm and Architecture." IEEE Proceedings of the 1982 International Conference on Parallel Processing, pp. 289-300.

Chu, Wesley W.; Holloway, Leslie J.; Lan, Min-Tsung; and Efe, Kemal. "Task Allocation in Distributed Data Processing." Computer, November 1980, pp. 57-69.

Coffman, Edward G., and Denning, Peter J. Operating Systems Theory. Englewood Cliffs, N.J.: Prentice Hall, 1973.

Coffman, Edward G. (ed.) Computer and Job-Shop Scheduling Theory. New York: Wiley, 1976.

Special Issue on Supersystems for the 80's. Computer, November 1980.

Efe, Kemal. "Heuristic Models of Task Assignment Scheduling in Distributed Systems." Computer, June 1982, pp. 50-56.

Hockney, R. and Jesshope, C. Parallel Computers. Bristol: Hilger Ltd., 1981.

Kartashev, Svetlana P., and Kartashev, Steven I. "Adaptive Assignment of Hardware Resources for Dynamic Architectures." Parallel Computers. Bristol: Hilger Ltd., 1981.

Kartashev, Svetlana P., and Kartashev, Steven I. "A Distributed Operating System for a Powerful System with Dynamic Architecture." AFIPS Conference Proceedings, Vol. 51, 1982 National Computer Conference, Montvale: AFIPS Press.

Kartashev, Svetlana P., and Kartashev, Steven I. "Distribution of Programs for a System with Dynamic Architecture." IEEE Transactions on Computers, June 1982, pp. 488-514.

Kuck, David J. The Structure of Computers and Computations Volume I. New York: Wiley, 1978.

Kung, H. "The Structure of Parallel Algorithms." In Advances in Computers Vol. 19. 1980, pp. 65-112. Edited by Michael Yovits. New York: Academic Press.

Ma, Pern-Yi Richard; Lee, Edward Y. S.; and Tsuchiya, Masahiro. "On the Design of a Task Allocation Scheme for TCA." IEEE 1981 Real Time System Symposium, pp. 1022-1031.

Ma, Pern-Yi Richard; Lee, Edward Y. S.; and Tsuchiya, Masahiro. "A Task Allocation Model for Distributed Computing Systems." IEEE Transactions on Computers, January 1982, pp. 41-47.

Ma, Pern-Yi Richard. "A Model to Solve TCA Problems in Distributed Computing Systems." Computer, January 1984, pp. 62-68.

Padua, David A.; Kuck, David J.; and Lawrie, Duncan H. "High Speed Multiprocessors and Compilation Techniques." IEEE Transactions on Computers, September 1980, pp. 763-776.

Stone, Harold. "Multiprocessor Scheduling with the Aid of Network Flow Diagrams." IEEE Transactions on Software Engineering, January 1977, pp. 85-93.

Stone, Harold, and Bohhari, Shahid. "Control of Distributed Processes." Computer, July 1978, pp. 97-106.

Special Issue on Supersystems. IEEE Transactions on Computers. May 1982.

Vick, Charles R.; Kartashev, Svetlana P.; and Kartashev, Steven I. "Adaptable Architecture for Supersystems." IEEE Transactions on Computers, November 1980, pp. 17-35.

Ward, Mathew O. "The Automated Design of Task Specific Parallel Architectures." IEEE Proceedings of the 1982 International Confererence on Parallel Processing, pp. 298-300.