SCHEDULING ALGORITHMS FOR PARALLEL EXECUTION OF COMPUTER PROGRAMS

BY

FARIDEH ANSARI-JAFARI SAMADZADEH

Bachelor of Arts
College of Mass Communication
Tehran, Iran
1979

Master of Science
University of Southwestern Louisiana
Lafayette, Louisiana
1982

Master of Science
University of Southwestern Louisiana
Lafayette, Louisiana
1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY

July, 1992

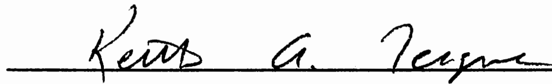
Thesis
1992D
S187s

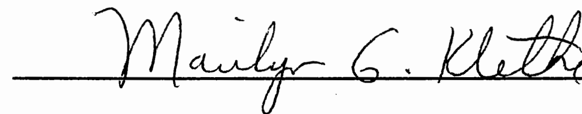
Oklahoma State Univ. Library

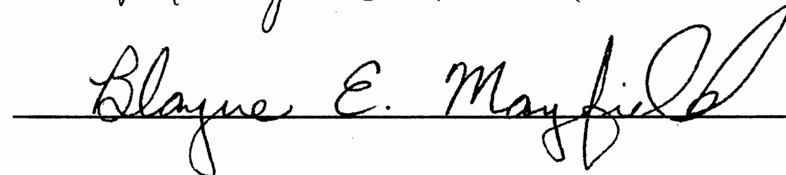
SCHEDULING ALGORITHMS FOR PARALLEL EXECUTION OF COMPUTER PROGRAMS

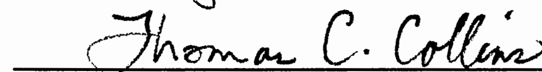
Thesis Approved:

Thesis Advisor









Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. G. E. Hedrick, for his trust in my work during the course of this research. I would also like to thank the members of my dissertation committee, Drs. Meg Kletke, Keith Teague, and Blayne Mayfield. I owe special thanks to Dr. Kletke for pointing out several helpful articles early in my research work.

I would like to express my most sincere appreciation and admiration to my mother, Ms. Esmat Haji-Ahmadi, who cheerfully devoted one whole year of her life to caring for my three daughters (Shahrzad, Nozlee, and Shereen) while I was completing my dissertation research. I have always been fortunate in having her full support in all my professional endeavors. Finally, I thank my best friend, colleague, and husband, Mansur, whose encouragements and deep trust in me was instrumental in completion of this research.

The future work on this dissertation topic will be partly funded by an NSF grant, with the author of this dissertation serving as the Principal Investigator.

TABLE OF CONTENTS

Chapter | Page

I. INTRODUCTION 1
- 1.1 Statement of the Problem 6
- 1.2 Scheduling Problem and NP-Completeness 9
- 1.3 Review of the Literature 10
- 1.4 Background and Survey of Related Work 12
- 1.5 Dissertation Overview 17
- 1.6 Notation and Definitions 18
- 1.7 Summary 21

II. PROGRAM PARTITIONING, TASK SYSTEMS AND INTERPROCESS COMMUNICATION
- 2.1 Basic Definitions 22
- 2.2 Multiprocessor Architectures and Task Granularity 23
- 2.3 Graph-Based Program Representations 24
- 2.4 Different Approaches to Program Partitioning 25
- 2.5 Communication Costs and Interprocess Communication 28
- 2.6 Summary 32

III. SCHEDULING OF TASK SYSTEMS
- 3.1 Basic Definitions 33
- 3.2 Task System Representations 34
- 3.3 Task Systems with Independent Tasks 35
- 3.4 Task Systems with Dependent Tasks 37
- 3.5 Summary 39

IV. SCHEDULING OF INDEPENDENT TASKS
- 4.1 Introduction 40
- 4.2 Survey of Related Work 43
- 4.3 A Near-Optimal Scheduling Algorithm for Scheduling of Independent Tasks 51

Chapter Page

4.4 Performance Evaluation 55
4.4.1 Design Methodology 55
4.4.2 Simulation Results 56
4.4.3 Complexity Analysis 58
4.5 Summary 59

V. SCHEDULING OF DEPENDENT TASKS

5.1 Introduction 61
5.2 Task System Graphs and Schedule Bounds 62
5.3 Task System Partitioning 65
5.4 An Algorithm for Task System Partitioning 66
5.5 ESP/VL: A Scheduling Algorithm for Scheduling Independent Task Layers 72
5.5.1 Performance Evaluation 75
5.5.1.1 Design Methodology 76
5.5.1.2 Simulation Results 77
5.6 Ranked Weight Algorithm: A Heuristic Approach for Scheduling Dependent Task Systems 78
5.6.1 The Ranked Weight Heuristic 80
5.6.2 The Ranked Weight Algorithm 83
5.6.3 Algorithm Description 86
5.6.4 Performance Evaluation 90
5.6.4.1 Design Methodology 91
5.6.4.2 Simulation Results 91
5.7 Independent Path Scheduler 94

VI. SUMMARY AND CONCLUSIONS

6.1 Introduction 99
6.2 Summary 99
6.3 Contributions 101
6.4 Future Work 102

REFERENCES AND SELECTED BIBLIOGRAPHY 105

LIST OF TABLES

Table Page

I. Results of Experiment I 57

II. Results of Experiment II 58

III. Experiments with varying the number of tasks 79

IV. Ranked weight and expected processing times of tasks in Figure 8 87

V. Results of 300 Test Runs where the Number of Processors P = Width of DAG 92

VI. Results of 300 Test Runs where the Number of Processors 2 <= P <= Width of DAG 94

VII. Characteristics of Several Multiprocessor Scheduling Algorithms 100

LIST OF FIGURES

Figure Page

1. A Sample Precedence Graph and a Sample DAG 38

2. The Cutting Stock Problem 42

3. Non-monotonocity of First-Fit-Decreasing Heuristic in packing of items into bins 47

4. Non-monotonocity of First-Fit Heuristic in packing of items into bins 48

5. Phase I reordering of D&F algorithm 50

6. The Variant-Load Algorithm 53

7. A schedule for task system on three processors 55

8. A task system precedence graph 63

9. A simple precedence graph 64

10. Earliest Schedule Partition (ESP) for the task graph in Figure 13 67

11. Latest Schedule Partition (LSP) for the task graph in Figure 13 68

12. The ESP Algorithm 70

13. A task system represented as a weighted directed acyclic graph 71

14. A schedule for the task system in Figure 13 on three processors 74

15. Performance of the ESP/VL scheduling approach 79

16. Task graph G' resulting from removal of task A in Figure 8 82

17. Task graph G'' resulting from removal of task C in Figure 16 83

Figure Page

18. The Ranked Weight Algorithm 85

19. Successor list (a) and predecessor list (b) 88

20. A schedule for the task system of Figure 8 89

21. Schedule for the modified version of the task system in Figure 8 89

22. Vertical Partitioning of the task system graph in Figure 13 96

23. A schedule for the task system graph in Figure 13 on five processors 97

CHAPTER I

INTRODUCTION

Technological advances have produced significant changes in the field of computing. There are indications that by the end of this millennium, fundamental limitations (e.g., the speed of light) will have been encountered in circuit design [Seiworek 89] [Lea 87]. To attain higher processing speeds, computations must be moved to machines with multiple processors in order to defeat these physical limitations. Developing software for such machines generally can be classified as parallel programming. Parallel programming imposes certain constraints on computations which must be dealt with by both the future machines and programs (languages).

The term parallel processing refers to the class of activities in which two or more processes execute on two or more processing elements simultaneously. It is possible for the involved processes to belong to different computations or to the same computation. In the former case, parallel processing helps only in increasing the overall throughput of the system. In latter case parallel processing allows for the realization of speed-up in the computation of a single program. Informally, parallelism can be defined as doing more than one thing at once. Other similar interpretations of parallelism are: performing n activities at once; carrying out one activity in n simultaneous parts; or doing n different activities staggered in a time frame [Desrochers 87]. The term "parallel events" is defined as the occurrence of events during the same interval, whereas the term "simultaneous events" is defined as occurrence of events at the same instant [Hwang and Briggs 84].

Parallel processing has been a topic of interest for decades. There exist in the literature proposals for parallel architectures as early as 1945 [Hockney and Jesshope 81]. Language aspects of parallel programming have been investigated also for over three decades [Conway 63] [Gill 58] [Dreifus 58]. The concept of parallelism is not new. John von Neumann initially envisioned a parallel architecture but abandoned the idea because of the unreliability and bulkiness of the technology of the time [Hockney and Jesshope 81]. Parallelism in practice is not a new topic either. Since the early 1970's, with the third generation architectures, additional processors (satellite processors or input/output channels) have been employed to improve the efficiency of the central processing unit. Later developments in parallel processing resulted in the introduction of array processors and vector processors and the employment of pipelining in these architectures to speed up computations further.

With the advent of local area networks in the 1980's, distributed environments provided higher availability and reconfigurability which proved to be a promising field for attaining high performance computing. It is anticipated that the very high performance environments of the future will employ and integrate distributed processing, array processing and multiprocessing in order to take advantage of the best of what each of these approaches have to offer [Glenbe 91]. Although Amdahl's law [Minsky 70] [Amdahl 67] presents arguments against large systems, the limitations of the architectural and physical ingredients must be taken into consideration. Some typical limitations include the speed of light, the distances that cannot be shorter than certain possible lengths, and the component sizes. Unless radical and new technologies such as optics or superconductors are realized, semiconductor technology is reaching its peak performance. Therefore, it is the multiprocessing capability that offers unlimited computing speed.

Advances in hardware design and subsequent availability of high-speed computing

power at reasonable cost seems to have accelerated the research in parallel programming to some extent. Unfortunately, the same trend observed in the late sixties, signified by a widening gap between rapid advances in hardware development and the relatively slow rate of progress in software development, can be observed today when one studies the progress in delivery of parallel processing hardware to the market and the software tools and techniques that can harness this new computing power. In fact, most parallel machines have greater capabilities than a programmer knows how to apply to a single concurrent program.

A review of the literature reveals arguments favoring each of the two architectural styles of uniprocessing and multiprocessing. Coffman analyzed the performance of multiprocessors from a queuing theory point of view [Coffman 66] [Coffman 67]. He concluded that the average number of tasks processed is greater for C processors than $C/2$ processors whose average execution speed is twice the average speed of each one of the C processors. That is to say, distributing the computing power over a larger number of processors yields a better throughput than over machines with half the number of processors with twice the speed. Grosch's law [Ein-Dor 85] asserts that a most powerful uniprocessor delivers the best price/performance. This law, however, is no longer true based on the influence of new advances in parallel processing [Lea 87]. Additionally, as physical limitations in the design of circuits are being approached, uniprocessor systems will no longer be able to respond to the demand placed on computers today. Other factors that make parallel processing a more attractive alternative are extensibility, productivity, reliability, and fault tolerance [Trelevin 90].

Some applications that are particularly suited for parallel processing are signal processing [Hwang and DeGroot 89], graph problems [Hirchberg 82], and scientific computations [Wilhelmson 87] such as three dimensional partial differential equation solutions [Peterson 85], Monte-Carlo techniques in physics and chemistry [Kalos 87],

and weather forecasting. Due to advances in technology and the realization of new architectures that allow for delivery of multiprocessors to the market using affordable off-the-shelf processing elements (e.g., Sequent and WYSE computers), availability of high-speed supercomputing-class performance is becoming a reality to those users who find supercomputers unaffordable, e.g., see [Karin and Smith 87]. Two of the most challenging problems in multiprocessing are the detection of parallelism in different portions of a program (program partitioning) and scheduling of the resulting parallel components on a number of available processors (mapping of computations to processors). This dissertation concentrates on the latter problem.

Multiprocessor scheduling has benefited greatly from the theoretical work done on the scheduling problem in other disciplines such as operations research and management science. What follows is a general discussion of the scheduling problem. More specific and targeted discussions of the scheduling problem, related to multiprocessor scheduling are presented in Section 1.4. Depending on the field of research, the term *scheduling* refers to different activities. For example, in operations research, there is a distinction between sequencing and scheduling [Noronha and Sarma 91]. The term sequencing refers to ordering of events without any reference to time constraints while scheduling involves specifying the exact start and finish times of operations.

In general, scheduling can be defined as the process of designing a procedure for sequencing a set of desired activities that take place over a set of objects. This sequencing is based on the time constraints for the delivery of the results, or availability of the resources necessary for performing an action. Different taxonomies can be defined depending on the type and the number of machines available and also based on the type of jobs that must be scheduled on the given machine(s). The questions that generally must be answered are:

1) How many machines are available?

2) Are the machines identical in their capabilities?

3) Are there any precedence constraints among the jobs that affect the processing sequence of these jobs?

The collection of answers to each of these questions defines the particular configuration of a system. The first question is aimed at determining whether the system is a single- or multiple-machine system. The answer to the second question reveals the interrelationship between the different machines. For example, if the system is composed of machines with different capabilities, we are probably dealing with a flow-line production system where each job possibly will go through every single machine (e.g., assembly lines and pipelining). However, if the system is composed of machines with identical capabilities, then unit jobs probably will complete on the same machine that they are assigned to. In this situation, we are dealing with a parallel-machine problem aimed at increasing the system throughput if the unit jobs are independent, or increasing the execution speed of the same job if the unit jobs are related and are in fact tasks constituting a single job. The answer to the third question reveals the interrelationships among the jobs to be scheduled. If there are no precedence relationships among the jobs, the sequencing of the jobs is done with regard to the delivery constraints and due dates (if any). In the absence of due date constraints, the overall performance of a schedule is measured by average turnaround time in the case of a single-machine problem, and minimum makespan (schedule length), and/or turnaround time, in the case of parallel machine problems.

Precedence relationships between pairs of jobs impose constraints on the order in which the jobs can be scheduled on machines. This problem is easier to deal with in a single-machine system than a multiple-machine environment. Typically, the complexity of devising an *optimal* and even a *reasonable* schedule increases exponentially as the number of machines and the jobs increase. Further discussion about scheduling

complexity is deferred to Section 1.2.

The discussion so far concentrated on the general problem of scheduling jobs on machines. This discussion could be applied to the problem of scheduling in many different fields such as job-shop scheduling [Coffman 76], project scheduling [Davis 73], mass-transit scheduling [Bodin 83], and multiprocessor and task scheduling [Coffman and Graham 72]. This research is concerned mainly with the scheduling of task systems on multiprocessors. However, a study of the problem of scheduling in other fields can be beneficial since there are many similarities between these problem domains. For example, bin packing algorithms applied to memory allocation and processor scheduling [Franklin 78] [Coffman et al. 78] were modeled after a more general problem known as the *cutting stock* problem in operations research [Gilmore and Gomory 61].

This dissertation is concerned with the scheduling of task systems on a computer with multiple identical processors. The task systems may contain precedence constraints. Heuristics are presented for scheduling of task systems on multiprocessors with the objective of creating schedules with near-optimal makespans. We define a makespan or schedule length to be the elapsed time between the time the first task is scheduled and the time the last task is completed.

1.1 Statement of the Problem

The main and most challenging task of multiprocessor systems is to increase the speed of execution of individual programs. An implicitly lower priority task, of course, is increasing the system throughput by keeping the processor utilization at maximum level. One way to increase the utilization is to increase the number of concurrent users on the system that run non-concurrent applications under time-sharing. However, this solution may increase the utilization but it probably will adversely impact the individual turnaround times because of the contention for system resources. Such users may be

better off on a serial time-shared machine. Therefore, parallel processors, arguably, are best suited for attaining higher speed for execution of individual jobs.

In order to speed up the execution of programs or in other words, to minimize the total execution time required for processing a single job, different portions of a program must be assigned to different processors and executed concurrently. Each of the identified portions in such a program are referred to as a *task*. The problem of assignment of tasks to processors is referred to as *mapping*. We refer to the process of mapping and sequencing of tasks on processors as *scheduling*.

One of the most critical issues in extracting good performance from a multiprocessor system is the scheduling mechanism used. We would like the scheduling scheme to be such that it devises schedules that minimize the total execution time of a program. Factors that affect the total execution time (schedule length) of a program, other than the balance in the individual workloads assigned to different processors, include synchronization and interprocessor communication costs. We refer to such costs as overhead, collectively. An additional overhead involves the processing time required by the scheduler itself in order to devise a schedule for a program. As discussed in the next section, the problem of scheduling of tasks on processors is a combinatorial optimization problem and is therefore, NP-complete. Because of this fact, research endeavors concentrate on sub-optimal or approximation algorithms for solving the scheduling problem by using heuristics.

As discussed in the section on review of related work for chapters 4 and 5, some of the existing scheduling algorithms incur a high overhead in devising schedules and therefore create a bottleneck for the system. Ideally, we would like a scheduling algorithm to have a low run-time complexity and at the same time, produce good schedules. In order to execute a program on several processors, a scheduling mechanism must assign different tasks in a task system (the representation of a program after partitioning of the program)

to different processors. The problems faced in scheduling of the tasks in a task system involve i) scheduling the tasks in such an order that correct execution sequence (arising from dependencies between pairs of tasks) is guaranteed, and ii) balancing the workload among the available processors such that the total execution time for a program is minimized.

This research concentrates on developing scheduling algorithms that are suitable for scheduling of dependent and independent task systems. Formal definitions of dependent and independent task sets are presented in chapter 3. Informally, an independent task system (or task set) is a task set in which individual tasks comprising the task set, do not exhibit any communication or data dependencies. On the other hand, dependent task systems are defined to contain tasks that exhibit communication, control, or data dependencies. Obviously, when scheduling dependent task sets, dependency constraints will have to be taken into consideration in order to satisfy the *determinacy* criteria [Coffman and Denning 73]. The determinacy property can be defined as follows. It is an accepted fact that in an operating system environment, no assumptions can be made about the relative speed of execution of different processors. Additionally, different partial orders may exist for execution of a given set of tasks. Given these two parameters, a task system is said to be determinate, speed-independent, or functional, if the uniqueness of results is guaranteed regardless of the partial execution order or the speed of execution.

Satisfying the determinacy property is not a concern in scheduling of independent task sets. Therefore, the sole objective in scheduling of independent tasks is producing the shortest possible schedule length since the tasks in such a task system are mutually non-interfering. However, scheduling of dependent tasks is concerned with the same objective of producing shortest possible schedule lengths, with the additional constraint of satisfying the determinacy problem. The purpose of the current research is to develop

multiprocessor scheduling algorithms that address and solve the above problems.

As discussed earlier, the overall goal of multiprocessor scheduling is devising schedules that produce the shortest possible schedule length. Task system characteristics as well as the particular architecture used for execution of programs affect the schedule length. This dissertation research investigates different task system topologies and present scheduling algorithms that are sensitive to the task system topology in order to produce near-optimal schedules. The suitability of the developed algorithms for different architectures is addressed and discussed as well.

1.2 Scheduling Problem and NP-Completeness

Multiprocessor scheduling, in the context of this dissertation, can be defined as the scheduling of a set of n tasks on p independent and identical processors. The execution sequence of the tasks may be constrained by certain precedence relations. The objective is to devise an assignment of tasks to processors, considering the precedence constraints, such that the overall execution length of the task system is minimized. Assignment and sequencing of tasks on processors is referred to as a *schedule*.

The scheduling problem has a seemingly simple and straightforward solution in which every possible input sequence must be examined. The solution is the input sequence that optimizes the objective function (i.e, the shortest possible schedule length) while satisfying the constraints of the problem. Unfortunately, such an enumeration in search of an optimal schedule is not feasible in general since the computation time required grows exponentially as n and p grow, where n is the cardinality of the input set and p is the number of available processors.

The dominant factor in the complexity of the scheduling problem is the input size n, the number of tasks to be scheduled. An algorithm is said to work in polynomial time if the complexity of the algorithm is a polynomial in n. If the complexity of an algorithm

is exponential in n, the algorithm is said to work in exponential time which results in classifying the algorithm as an NP-complete algorithm, indicating that the algorithm may yield a solution in non-polynomial time. The general problem of scheduling requires computational time that grows exponentially with the number of tasks in the task system and thus is known to be NP-hard [Ullman 67]. Classifying a problem to be NP-hard means that it is as difficult to solve as the hardest problem that belongs to the NP family.

In light of the fact that optimal schedules cannot be found within a reasonable time frame, many research endeavors concentrate on finding near-optimal solutions in polynomial time. From among polynomial time solutions, those with relatively slower growth rate (as the input size increases) are rated to be superior to other cases (one such algorithm developed in this dissertation research is presented in Section 5.6). We are interested in developing scheduling algorithms that yield optimal solutions for a subset of the general problem domain and reasonable solutions for others in polynomial time.

1.3 Review of Literature

The two most important problems of interest that allow for the efficient use of parallel processing power, can be identified as the detection of parallelism in computer programs and the scheduling of the resulting parallel tasks on a target machine. Both of these problems have been the topic of numerous research endeavors since the early sixties. However, most of the earlier work concentrated on the first problem (i.e., that of parallelism detection). This was probably due to lack of widespread availability of multiprocessor machines in the 1960s and 1970s.

However, some research was carried out in the area of measurement of the performance of parallel processing machines and scheduling techniques that minimize the total execution time of a task system on a multiprocessor. In some of these studies, analytical investigations were performed to measure the overall performance of

multiprocessors in terms of the system throughput. Coffman analyzed the performance of multiprocessors from a queuing theory point of view [Coffman 66] [Coffman 67]. McNaughton and Rothkopf [McNaughton 59] [Rothkopf 66] and several other researchers investigated the problem of multiprocessor scheduling using graph model analysis techniques. All these models represent programs as directed acyclic graphs (DAG) and use such attributes as time, the frequency of execution of tasks, and their deadlines and penalties.

Hu's algorithm, famous for the optimal schedule it yields, was devised for scheduling of task systems represented as a tree [Hu 61]. Coffman and Graham devised an optimal scheduling algorithm for a two-processor system [Coffman and Graham 72]. their algorithm requires that the tasks in a task system have equal execution times. Also, the fact that the optimality of the resulting schedule is guaranteed for two processors only, seems restrictive even in the absence of computers with massive number of processors [Waltz 87]. Decomposing a program into equal-sized tasks for Coffman and Graham's algorithm is a major concern because the general problem of program decomposition into tasks for parallel execution has been proved to be recursively unsolvable [Bernstein 66].

Because of the NP-completeness of the general problem of scheduling, most sub-optimal scheduling algorithms use heuristics for producing near-optimal schedules (for a general survey of heuristic techniques see [Noronha and Sarma 91]). Some of the more well-known task scheduling algorithms in this genre are Longest-Processing-Time (LPT) [Graham 69] [Graham 76], Multifit [Coffman et al. 78], Divide & Fold (D & F) [Polychronopoulos 86], and CP/MISF [Kasahara and Narita 84].

Another class of scheduling algorithms is known as *list scheduling* [Coffman and Denning 73]. In a list scheduling algorithm, the tasks in a task system are sequenced in what can be described as a list based on a partial ordering. The partial ordering is achieved through imposing some type of *priority scheme* on the tasks. A list scheduler

operates by scanning the list and selecting tasks that satisfy certain priority constraints or urgency rules. Some of the urgency rules used in scheduling research are [Baer 73]:

- Smallest processing time first;
- Largest processing time first;
- Greatest number of immediate successors first; and
- Greatest number of successors first.

Unlike earlier work that concentrated on program partitioning and general performance issues of parallel processing, recent research endeavors pay attention to other aspects of parallelism. Some current issues include developing optimal parallel algorithms [Guan and Langston 91], loop parallelization and scheduling [Saltz et al. 91], time-cost analysis of parallel computations [Qin et al. 91], graph-based partitioning and concurrency analysis [Long and Clarke 89] [Girkar and Polychronopoulos 88] [Agrawal and Jagadish 88] [Ammarguellat 90], measurement of parallel processor performance [Karp and Flatt 90], scheduling and task dependencies [Shang and Fortes 91], Hardware concurrency extraction [Uht 91], parallelizing compilers and environments [Kuck et al. 86], parallel processing in distributed environments [Feitelson and Rudolph 90], and a number of other areas such as attempts at developing software tools for mapping computations to architectures [Lo et al. 90], and automatic detection and parallelization of programs [Terrano et al. 89] [McGreary and Gill 89].

1.4 Background and Survey of Related Work

General characteristics of scheduling schemes can be defined in terms of the following properties. A scheduling scheme is either exact or heuristic, uses either a static or a dynamic approach, and is performed either on a preemptive or non-preemptive basis. Discussion of the choice of exact and heuristic approaches was presented in Section 1.2, "Scheduling Problem and NP-Completeness". There are arguments in the literature

supporting both static and dynamic approaches. There are two major arguments against dynamic scheduling. The first argument states that since compile time information and general topology of the task system graphs are not used in the dispatching of tasks, the selection of tasks may not lead to the best possible choice. The second argument concentrates on the run-time overhead of dispatching which can be quite high. These arguments lead to two more favorable choices. One choice is quasi-static scheduling in which compile time information is used to produce a preliminary schedule, which is then adjusted through processor synchronization if the actual processing times vary from earlier estimates [Ha and Lee 91]. The other choice is auto-scheduling where compile time information is used for sequencing of task executions [Polychronopoulos 88]. The arguments against fully static scheduling are predicated on the fact that the compiler produces schedules based on estimates and that realistic schedules are not always possible except in cases where all necessary information is readily available at the time of scheduling (e.g., systolic arrays).

A preemptive scheduling approach generally yields better results than a non-preemptive approach from a theoretical point of view. A preemptive schedule is more likely to yield an optimal schedule length. Under this approach, the idle time created between the scheduling of two tasks can be filled with running a portion of a ready task and therefore giving the scheduler a greater degree of flexibility in the scheduling of the tasks. However, in practice, preemptive scheduling has its own disadvantages, namely, incurring the overhead of context switching.

McNaughton [McNaughton 59] was the first researcher to introduce parallel machine scheduling. He introduced a scheduling algorithm that performed a preemptive scheduling scheme on a set of jobs. He defined the minimum makespan for a set of preemptive jobs as $\max\{t_1, t_2, ..., t_n, \sum_{1 \le i \le n} t_i/p\}$ which is known as the McNaughton lower bound. McNaughton introduced three performance criteria for parallel execution

of jobs. He used the lower bound mentioned above as a measure for scheduling of preemptive jobs. He refers to this type of scheduling as completion time based scheduling (CTB). He showed that his algorithm produces optimal schedules with a maximum of $p-1$ job preemptions, where p is the number of available machines. The other two performance measures defined are due-date based (DDB) and flow-time based (FTB) measures. In the DDB performance measure, the total weighted tardiness of the jobs is the performance measure. In the FTB measure, the due-dates are equal to zero and therefore, the flow time of the jobs is used as the performance measure. Most of the other approaches developed by researchers can be defined to belong to one of the three categories defined by McNaughton as discussed above.

Because of the NP-completeness of the general problem of scheduling, most known optimal solutions owe their optimality to rigid conditions imposed on the problem. Three of the well-known scheduling algorithms yielding optimal solutions are due to Hu [Hu 61] , Coffman and Graham [Coffman and Graham 72], and Muntz and Coffman [Muntz and Coffman 69]. All three of these algorithms are completion time based algorithms. Hu's algorithm operates on a special class of graphs where the precedence relations define a directed singly-rooted tree in which (except for the root vertex that has a fan-out degree of zero) each vertex has a fan-out degree of exactly one. Additionally, the tasks (vertices) are assumed to have unit or uniform execution times. The urgency rule (the priority scheme defined for the selection of tasks) used in scheduling of tasks on the next available processor causes a task that has no predecessors and lies on a path with the longest distance from the root vertex to be selected. Hu's algorithm runs in $O(n)$. Hu's method is referred to as the *highest level first* approach or the *critical path* method. Coffman and Graham's algorithm operates on directed acyclic graphs (DAG) with general precedence relations. They employ Hu's highest level first approach for selection of the tasks for scheduling. The limitations of this algorithm lie in the fact that, analogous to Hu's algorithm, the tasks must have unit execution times and that optimality

of the schedule is guaranteed only with two processors. Coffman and Graham's algorithm has $O(n^2)$ complexity. Muntz and Coffman's algorithm uses a preemptive approach and operates on arbitrary task systems (e.g., independent tasks, tree graphs, and DAGs). Their algorithm requires that tasks have *commensurable* execution times. This algorithm produces an optimal schedule of length $T_{OPT} = \max\{\max_{1 \le i \le n} \{w(t_i)\}, 1/p \sum_{i=1}^{n} w(t_i)\}$ where $w(t_i)$ denotes the processing time of task t_i. It schedules the n tasks with pn preemptions and runs in $O(n^2)$. This algorithm does not guarantee optimality for $p > 2$.

There are numerous algorithms and approaches that are based on Hu's basic idea of highest level first approach. Motivated by Hu's approach, Graham [Graham 69] introduced a new approach for task scheduling known as *list scheduling*. Many scheduling algorithms in the literature, which use urgency rules for selection or dispatching of tasks, can be categorized as list scheduling algorithms. In list scheduling, tasks in a task system are sequenced in what can be described a list based on a partial ordering. The assignment of tasks to processors takes place by scanning the list and selecting a task that satisfies the defined urgency rule for dispatching of tasks. Some typical urgency rules are largest processing time first (LPT), shortest processing time first (SPT), and greatest number of immediate successors first.

Certain anomalies and bounds related to list scheduling [Graham 69] are listed below (item 5 below, is a new anomaly discovered in this dissertation research).

1. There exist cases where no list scheduling scheme results in an optimal schedule for a given number of processors.
2. The reduction of task weights may result in increased schedule length.
3. Allocation of a larger number of processors may result in an increased schedule length(an example of this anomaly is shown in Figure 3).

4. A decrease in the number of arcs in a graph (i.e., relaxing the precedence constraints) may result in an increase/decrease in the schedule length.

5. Switching the execution times of tasks in the same task graph (without changing the total processing time or the topology of the graph) may result in an increase/decrease in the schedule length (e.g., refer to the schedules for the task graph of Figure 8, in Figures 20 and 21).

Graham proves that in the worst case, list scheduling yields schedules that are twice as long as the optimal schedule.

It was discussed earlier that scheduling schemes are either completion time based (CTB), due-date based (DDB), or flow-time based (FTB). FTB performance measures are discussed in this dissertation. The reader is referred to a state-of-the-art review of machine scheduling by Cheng and Sin for DDB and CTB schemes [Cheng and sin 90]. Other survey studies done in the area of multiprocessor scheduling are those by Gonzalez[Gonzalez 77] and Baer [Baer 73].

Gonzalez performed a comprehensive survey of deterministic scheduling of jobs in uniprocessing, multiprocessing, and job-shop environments. He presents classification categories for characterizing scheduling algorithms for different environments that are discussed in his survey. The surveyed research work are evaluated in terms of the number of processors employed, task execution lengths, precedence graph structures, task interruptibility, introduction of processor idle times into scheduling of individual tasks, presense or absence of deadlines for delivery of results, homogeneity or heterogeneity of processors, and the boundedness of available resources necessary for the execution of jobs.

The measures of performance used in Gonzalez's survey are such measures as minimization of finishing or completion times, minimization of the number of required

processors, minimization of mean flow times, maximization of processor utilization, and minimization of processor idle times.

The survey done by Baer concentrates on the theoretical aspects of the general problem of multiprocessing. Baer's survey discusses early language features suitable for explicit exploitation of parallelism such as fork-join [Conway 63] [Denning 66], and loop parallelization [Gosden 66]. This survey also discusses representations used for modeling of parallel computations. These models include different graph models such as directed graphs, directed acyclic graphs, flow graphs and their extensions, and Petri net-based models. Baer's survey also presents some discussion on different performance evaluation techniques such as queuing theory analysis techniques and simulation and modeling techniques.

More specific and targeted discussion of related work is presented in appropriate chapters.

1.5 Dissertation Overview

This dissertation is concerned with developing algorithms for scheduling of task systems on multiprocessors. Several different multiprocessor scheduling algorithms and approaches have been developed for scheduling of dependent and independent task systems. Performance of the developed algorithms has been compared to the performance of the best-known algorithms in the literature through simulation studies. The outline of this dissertation is as follows. The remainder of this chapter discusses the basic concepts, notation and the definitions used through out this dissertation.

Chapter two discusses the issues of task granularity as affected by the implementation platform. This chapter also discusses different approaches used for detection of parallelism in computer programs. Different models for task system representations and communication and synchronization costs are also presented and discussed.

Chapter 3 concentrates on the topic of task system characteristics. Formal and informal definitions of dependent and independent task systems are given. Bounds on the schedule lengths for each of the two classes of task systems mentioned above are presented and discussed.

Chapter 4 is devoted entirely to the issue of scheduling of independent serial tasks. A survey of related work is presented. A new algorithm for scheduling of independent serial tasks developed in this dissertation research, is presented in Section 4.3. The performance of this algorithm is compared to three of the best known algorithms. It is shown that our algorithm does at least as good as the best known algorithms and has a run-time complexity that is significantly better than those of the evaluated algorithms.

Chapter 5 concentrates on the scheduling of dependent task systems. Three different approaches and algorithms that address different needs in different environments have been developed in this dissertation research. The suitability of the developed approaches in relation to the task system and the architectural characteristics are discussed. Performance of the developed algorithms has been measured through simulation studies.

Chapter 6 presents a summary of the results and the conclusions reached in this research. Future work with respect to the current research is also discussed.

1.6 Notation and Definitions

In this section, the notation used in the discussion which follows is introduced. Given n independent, tasks each task is denoted by t_i, $1 \le i \le n$. Therefore, the task system can be represented as a set, $\tau = \{t_1, t_2, \ldots, t_n\}$. Associated with each task t_i is a non-negative integer value $w(t_i)$ which is the weight (normally, the execution time) of that task. Henceforth, $w(t_i)$ is referred to as the size, the weight, or the execution time of a task, interchangeably. The sequential execution time of task system τ, denoted by T_s, is defined to be

$$T_s = \sum_{i=1}^{n} w(t_i)$$

We are interested also in the time it takes to execute a given task system in parallel. This value is represented by T_p. We refer to T_p as the length of a schedule or execution length. A specific formula for T_p cannot be derived since it depends on the relative sizes of tasks, the particular scheduling algorithm used, the number of processors, and the nature of processing in the execution platform. However, some upper and/or lower bounds on the lengths of the schedules in various situations are presented in the subsequent chapters.

Given a task system and a number of processors, we are interested in finding an optimal schedule length for the given task system. However, it was discussed in Section 1.2 that determining an optimal schedule for the general case in polynomial time is not possible. We denote an optimal schedule (if one exists) as OPT and denote its length as T_{OPT}. In order to measure the performance of different scheduling algorithms, we need to know the optimal schedule length for a given problem. Since T_{OPT} is not known for a general problem, we will use an approximation of the optimal schedule length and denote it as T^*_{OPT}. Different approximations of T_{OPT} will be discussed in appropriate sections. Through out this dissertation, T_{OPT} is referred to as an optimal schedule length if one is known or its approximation T^*_{OPT}.

We define a *path* through a graph to be a traversal through a sequence of nodes that starts at the first node (the source node or source) and leads to the last node of the graph (the sink node or sink). A *critical path* in a graph is a path that has the largest cumulative processing time, $w(t_i)$, of the nodes, t_i, visited in such a traversal. We denote the length of a critical path by T_c. It should be noted that given a DAG, $T_{OPT} = T_c$; that is, given an unlimited number of processors, no scheduling algorithm can yield a schedule shorter than T_c.

In the discussion that follows, we are interested also in the speed-up S_{τ} achieved for a task system τ, when executing that task system on several processors. This will be calculated using the formula

$$S_{\tau} = T_s / T_p$$

The highest possible speed-up in parallel execution of the task system τ is denoted by $S_{\max}$ and is calculated as

$$S_{\max} = T_s / T_{OPT}$$

where T_{OPT} stands for the optimal schedule length or an approximation of the optimal schedule length T^*_{OPT}, defined through some bounds.

As discussed in Section 1.2, the scheduling problem is in effect, a combinatorial optimization problem and is therefore, NP-complete. As a result, most practical scheduling algorithms that exist in the literature are approximation algorithms. In order to measure the performance of an approximation algorithm, the performance ratio is defined as follows [Coffman et al. 78].

$$R_p(A) = T_A / T_{OPT}$$

where T_A is the performance of an approximation algorithm A, for a given problem, and T_{OPT} is the measure of performance for an optimal solution known for the same problem.

Unless specified otherwise, the particular architecture assumed in the discussions that follow is a shared memory multiprocessor machine in which all processors are identical in their capabilities.

(Other notation used in more targeted and specific discussions is introduced where appropriate.)

1.7 Summary

Chapter 1 presents a broad overview of the significance of parallel processors. Arguments that establish the prevailing choice of employing a large number of (less powerful) processors as opposed to a single powerful processor are presented. Mapping of computations to processors is identified as one of the most critical issues in improving the performance of parallel processors with respect to minimizing the execution time of individual programs. The statement of the main problem addressed in this dissertation is presented in Section 1.1. Section 1.2 presents arguments for justifying the practice of using approximation algorithms for solving the scheduling problem as opposed to seeking exact optimal solutions. A general review of the scheduling problem and the early theoretical work as well as current directions in parallel processing are presented in Sections 1.3 and 1.4. Section 1.5 presents an overview of the dissertation. Notation and definitions used throughout the remainder chapters are presented in Section 1.6.

CHAPTER II

PROGRAM PARTITIONING, TASK SYSTEMS AND INTERPROCESS COMMUNICATION

2.1 Basic Definitions

To speed up the execution of a program, portions of the program that potentially can be executed in parallel must be identified. This activity is referred to as partitioning. Different approaches used for program partitioning are discussed in Section 2.4. Each of the resulting blocks in such a partitioning is called a task. A task is defined as a set of instructions that once assigned to a processor, is executed sequentially. Furthermore, a set of clearly defined input and output parameters are associated with each task. Input dependencies determine the execution sequence of the tasks. Therefore, a task cannot be selected for execution until all of its predecessors have finished execution and thus deliver their output parameters. We show in chapter 5 how such a constraint can serve as the synchronization mechanism in scheduling of tasks under a variety of scheduling schemes proposed and examined in this dissertation.

The collection of identified tasks in a program is referred to as a task system. Generally, task systems are represented through graph models. Some example representations of task systems can be found in [Ramamoorthy 66] [Estrin and Turner 63] [Girkar and Polychronopoulos 88] [Long and Clarke 79] [Agrawal and Jagadish 88]. A more detailed examination of graph models is presented in Section 2.3.

A task system can be defined as the product of the application of a partitioning scheme to a program. A partitioning scheme can be viewed as an equivalence relation R, that

divides the program into a number of equivalence classes. Some example equivalence classes in a task system are procedures and functions, basic blocks, loops, and code segments that are separated from the rest of the program by branching actions. The equivalence relation defined for the partitioning of programs depends on the desired degree of granularity. Program granularity is defined as the relative size of the identified tasks in a program. Determining the degree of granularity is itself dependent on the machine characteristics.

2.2 Multiprocessor Architectures and Task Granularity

Program partitioning and task granularity are two closely related terms. Task granularity refers to the relative size of the identified tasks after the application of a particular partitioning scheme to a program. Task granularity is affected by different objectives and constraints. Depending on these objectives, one might define each single statement in a program to be a task. We refer to such a partitioning as the finest-grained partitioning (of course, in some cases, for example, in data flow architectures, partitioning can be done at an even finer grain). At the opposite end of this spectrum, one might define an entire program to be a task.

For the purpose of relating task granularity to machine characteristics, we will divide the machines based on their processor interconnection and communication schemes. Using the criterion of interprocessor communication, machines can be divided into two broad classes of loosely coupled and tightly coupled machines. In general, task granularity is coarser for loosely coupled or private memory machines, and may be finer for tightly coupled or shared memory machines. Therefore, in scheduling of jobs on a target machine, further consideration might warrant altering the degree of granularity of programs. For example, if a program is to be run on a machine with distributed memory on which communication between processors is proven to be costly, a coarser-grained partitioning might be desirable. On the other hand, shared memory machines allow for

finer-grained partitioning schemes. Regardless of the particular implementation platform, determining the degree of task granularity is affected by the communication cost involved in processing of input and output parameters, synchronization, and context switching costs.

2.3 Graph-based Program Representations

The majority of past and current work done in the area of detection of parallelism are based on graph models. Typically, these studies use as their first step, the control or data flow graph of a program. The control flow graph (which is most probably a directed cyclic graph due to the existence of loops) is then converted to a directed acyclic graph (DAG) using some loop removal algorithm (e.g., see [Ramamoorthy 66]). In order to convert a cyclic graph to a DAG, the boolean connectivity and precedence matrices of the graph must be created (for example by using Warshall's algorithm [Warshall 62]). Such models may attempt further to identify the input/output dependencies between pairs of statements using input and output matrices [Russel 69], to expose parallelism in loops [Volansky 70] [Muraoka 71], or to use the resulting DAG and concentrate only on those portions that can potentially be optimized and are amenable to parallel processing [Gonzalez and Ramamoorthy 71]. Other research work [Regis 72] [Keller 70] concentrate on using DAGs and utilizing the determinacy properties of graphs to represent a maximally parallel structure, or to create directed acyclic bilogic graphs (d.a.b) to represent the information structure of a graph by attaching labels such as "OR", "XOR", "FORK", etc., to the branches in a graph [Dennis 68] (d.a.b which is also known as the UCLA graph model [Estrin and Turner 63] is the basis for a number of other studies).

Some more recent studies [Skedzielewski and Glauert 85] that use flow graphs in their analyses, use an intermediate form of data flow graphs which is subsequently used by a compiler for machine code generation. Other studies use task interaction graphs to

represent concurrency of execution [Long and Clarke 89] or program dependence graphs for optimization and detection of parallelism and partitioning [Ferrante et al. 83], [Kuck et al. 80], [Burke and Cytron 86], [Wang and Gannon 89], [Wolfe 82], and [Girkar and Polychronopoulos 88], or use control dependence graphs [Sarkar 91] for identifying non-loop parallelism (e.g., IBM's PTRAN).

2.4 Different Approaches to Program Partitioning

General-purpose parallel machines (non-SIMD machines according to Flynn's Taxonomy [Flynn 72]) in general can serve two broad categories of "competing" and "cooperating" computations. Competing computations can be defined as unrelated computations that compete for the resources of a system. Parallel machines only help in increasing the throughput of the system in such circumstances. The second category is that of cooperating computations in which portions of the same job are assigned to different processors to speed up the computation of the same job. This dissertation is concerned with the latter category of computations.

In breaking a given job into parallel subtasks, two different approaches can be employed: i) The program first can be written in the sequential (i.e., non-parallel) format using a conventional programming language. The program can be then passed through a parallelizing compiler to extract the potentially parallel operations, or ii) The programmer can use parallel programming constructs available in the languages supported by a particular machine in the process of software development.

The two approaches of using parallelizing compilers or employing parallel programming constructs explicitly built into parallel programs have their advantages and disadvantages. The advantage of using parallelizing compilers is the portability of the code, but the generated code might not be as efficient as the program written with the aid of parallel programming constructs [Emrath 89]. To direct such compilers in recognizing

parallelism, comment-like statements called "assertions" must be inserted into the code manually. The code generated by parallelizing compilers must be examined repeatedly by the programmer, new assertions must be inserted if the programmer identifies any portion of the code whose potential for parallelism was not recognized by the compiler. As a result, some portions of the code may have to be modified and the program then must be passed through the compiler again. These iterations must continue until the desired degree of parallelism is achieved. Such an undertaking can become very time consuming and laborious.

It should be noted that different compilers do not interpret assertions in the same way and therefore, even in the case of compiler-generated parallel code, portability is a problem. The lack of portability of parallel code is due to lack of standard languages as well as the existence of drastically different architectural styles and hardware features on parallel machines [Emrath 89]. Due to the absence of standard languages, researchers and developers extend their own favorite language or write their own support libraries that are compatible with their own hardware and take advantage of the machine instructions and support the specific features available on their machines. When a program that utilizes such hardware support features is ported to a different machine, the absence of counterpart support has to be remedied in software, which might yield a version that is even slower than its sequential version [Emrath 89]. However, because of the relative inexperience in developing parallel programs and the current absence of standardized programming languages and/or efficient compiler support, one should not allow the issue of portability to get in the way of advancing other aspects of the field such as the issues of partitioning and decomposition of programs for parallel execution, and scheduling considerations of the program partitions.

The second approach for programming parallel processors involves the use of explicit parallel programming constructs in programs. This type of explicit expression of

parallelism could be introduced at different levels. Some applications might call for parallelism at procedure level such as "buffering mechanisms" and "producers and consumers" types of problems.

In an earlier study by this author, procedure-level parallelism for an application of the producer-consume relationship was investigated [Samadzadeh 91]. The implementation platform was a Sequent S-81 with 16 processors. The particular problem studied involved a series of regular expression substitutions in a tri-buffer mechanism. The buffers used were bounded buffers and therefore synchronization was a major issue in solving the problem. Mutual exclusion had to be enforced in manipulating critical sections as well. Because of the particular nature of the question posed in this experiment, the problem could be solved most elegantly in a parallel processing environment. The purpose of this study was not the issue of execution efficiency and the speed up of computation, because in general, efficiency is a primary concern in computationally-intensive applications. The purpose of this experiment was to investigate the ease of programming and debugging of parallel programs by novice programmers and the overall comprehensibility of such applications.

In the experiment cited in the above paragraph, the unit of granularity in parallel execution was defined to be tasks represented as entire procedures which required synchronization and interprocess communication. Parallelism could be introduced at finer grains also in which the units of computation are composed of blocks of code that serve a particular purpose. This can be done by using parallel programming library support in which the units of computation could be blocks of code with clearly defined input and output. Subsequently, the identified, tasks in the resulting task system could be coordinated and executed using explicit parallel programming constructs. Another form of parallelism, which is not a topic of interest in this research, is parallelism at the statement level. At this level of parallelism, potentially parallel components of a single

statement are identified, each of which might be executed on a separate processor and the results combined. It should be mentioned however, that this is considered to be the lowest level of granularity which is handled by the compiler. One can imagine that manual techniques for this type of parallelism can be extremely tedious if not impossible.

The problem with the use of explicit parallel programming constructs is identifying disjoint subcomputations within a program. This is done mostly on a trial and error basis until the desired degree of parallelism (usually measured by the number of independent tasks that can be executed concurrently) is achieved. Programmers, for example, might use a profiler to measure a program's behavior [Kwan et al. 90]. These measurements can then be used as a basis for altering the granularity of the program in terms of the number of tasks present until the desired degree of parallelism is achieved.

Detecting parallelism in programs is at best an intuitive process. As a consequence, it seems that the less experienced a programmer is, the more serious the problem of introducing a reasonable degree of parallelism into programs becomes. Even in the case of more experienced programmers, there is more parallelism embedded in programs than eyes can see [Ramamoorthy and Gonzalez 69]. Availability of methods and tools that allow for detection of parallelism in code can therefore take the guesswork out of the process of program design.

2.5 Communication Costs and Interprocess Communication

It is argued that identifying the maximal degree of parallelism, i.e., isolating as many tasks in a program as possible, can potentially result in the greatest possible speed up. Of course Bernstein has shown that the general problem of partitioning programs into tasks is recursively unsolvable [Bernstein 66]. On the other hand, as the number of tasks in a task system increase, the overhead in scheduling of the resulting tasks increases accordingly, and probably not necessarily linearly. Additionally, the number of

concurrent tasks present at each level of the resulting task system or concurrency graph might exceed the number of available processing elements. If no further action in coarsening the degree of granularity is taken, communication overhead and the overhead of scheduling of the many tasks present in the program will result in degrading the performance of the given program. Consequently, further restructuring of the concurrency graph will be necessary in order to adjust the degree of parallelism for a target machine.

Kwan, Bic, and Gajski [Kwan et al. 90] evaluate, in an experimental study, the improvement of the performance of parallel programs through the use of critical paths in a program. In their experiment, they define two measures, the data flow graph critical path and the scheduled critical path. They argued that the data flow critical paths represent the maximum possible parallelism available in a program. Using the data flow graph, they came up with the scheduled critical path which consists of some of the tasks present in the data flow graph. These tasks are assigned to a fixed number of processors. The scheduled program now has its own critical path which might be different from that of the control flow graph critical path because of the restructuring of the identified tasks in scheduling, based on their dependencies. It is possible that both critical paths are identical.

Each task on a scheduled program critical path is a unit of execution with clearly defined input and output. The input/output of tasks are used as means for determining necessary synchronization between tasks. Kwan et al. define a scheme for assigning weights to the scheduled program critical paths. The weight of a node (task) on a path is defined to be equal to the amount of processor time necessary to complete the task. Edges that connect tasks (and represent flow of information) also have a weight associated with them. The weight of an edge corresponds to the communication overhead and consists of the amount of time required to access an input/output parameter. The

overall weight of a critical path is the sum of the weights of all nodes and edges that lie on that path. The computational weight of the critical path is the sum of the weights of the nodes on a critical path. The authors use these weights (critical path weight, which includes the communication overhead, and the computational critical path weight, which does not take into consideration the communication overhead) to evaluate the performance of related tasks when assigned to different processors (as opposed to being scheduled on the same processor) to see if the communication overhead dominates the computation time for each critical path in this type of scheduling.

In this experiment, Kwan and his colleagues define a parameters P_{opt} as the optimum number of processors to achieve the highest speed up. Since determining P_{opt} could be an intractable problem [Bernstein 66], Kwan et al. define P_{mm} as an approximation of P_{opt} which represents the minimum number of processors for achieving maximum speed up. They designate P to represent the number of available processing elements. They then perform a series of experiments on solutions to the Gaussian elimination of a set of algebraic expressions and test the solutions with various degrees of granularity. The dependent variable in these series of experiments is the length of the critical paths on the scheduled programs. As they perform these experiments, they collect data about the weights of the scheduled critical paths and the computational weights of the critical paths. The collected data is used to test the effect of the level of granularity of tasks on the critical paths and the communication overheads of scheduling interdependent critical paths on different processors.

The results of these experiments on various degrees of granularity demonstrates a trend. Kwan et al. report that if the number of available processing elements P is much less than P_{mm}, refining granularity degrades performance. The reported reduction in performance is apparently due to the fact that when the number of concurrent tasks is larger than the number of available processing elements, more than one concurrent task

gets assigned to some processors, thereby, reducing the performance because of the context switch time associated with the scheduling of the related tasks. On the other hand, if $P \geq P_{mm}$, refining granularity results in execution speed up.

In general, the number of independent paths in a data flow graph represent the maximum possible parallelism. However, this degree of parallelism also involves the highest amount of overhead in the execution of tasks because of the communication dependencies among different tasks that might lie on different paths or that lie on the same path but are assigned to different processors. If one rearranges these tasks and schedules them on one processor, it is true that the time needed for communication is reduced (because tasks can communicate through local memory), but on the other hand as the degree of granularity is refined and the number of related tasks assigned to the same processor increases, there is increased tendency towards sequential execution instead of parallel execution because unrelated tasks that could potentially run on different processors are now assigned to the same processor. It should be noted that even if program segments are written as clearly defined and separate tasks, but are not assigned to different processors for concurrent execution, their execution proceeds in a sequential manner, even on a multiprocessor machine.

A significant finding in the experimental studies, done by Kwan and his colleagues [Kwan et al. 90], is that as granularity is refined and a program is divided into a larger number of tasks, it is true that the overhead in necessary communication between the related tasks increases (which is because each of a number of related tasks executes on a different processor), but the computation speed achieved by parallel execution of the related tasks offsets the communication overhead and furthermore results in overall program execution speed up. If the program designer is not bound by the availability of only a limited number of processors, then it is not the case that the finer the granularity, the higher the achieved speed up. Nonetheless, refining the granularity can lead to the

point of diminishing returns, in which the total execution time of a given task is less than or at best equal to the communication overhead involved between a given task and its related counterparts. It is also possible that the achieved speed up is not significant enough to justify the effort expended in taking advantage of the large number of processing elements that might be available on a target machine.

2.7 Summary

Chapter 2 discusses the issues related to the choice of task granularity as affected by the implementation platform. The two approaches of partitioning programs using parallelizing compilers (implicit parallelism) and using parallel processing constructs at the time of development of programs (explicit parallelism) are discussed. Overheads associated with the mapping of tasks to processors, such as interprocessor communication and synchronization costs, are discussed in the context of private memory and shared memory architectures.

CHAPTER III

SCHEDULING OF TASK SYSTEMS

3.1 Basic Definitions

We represent a program as a directed acyclic graph. Since the main focus of this dissertation is scheduling of a given task system on a multiprocessor, we do not concern ourselves with the details of producing the program graph. The program graph may be thought of as a graph representing the control flow of a program. Such information typically can be provided by a compiler. Additionally, it is assumed that the individual tasks (i.e., nodes in the program graph) are created using a bottom-up approach such that each task is created using "natural" boundaries (e.g., an outermost loop, a procedure call, or a basic block). Therefore, the resulting graph is indeed a DAG. A bottom-up approach is preferred to a top-down approach because of the following reasons. Synthesizing individual tasks, as individual statements are analyzed, yields faster results (by requiring fewer passes) and reveals dependencies between and among statements in a program better than starting the analysis from the top (outer) levels in a program [Polychronopoulos 86].

Task system scheduling involves the assignment and sequencing of tasks to processors. The length of a schedule is defined to be the elapsed time between the time the first task is assigned to a processor and the time the last task in a tasks system finishes execution. An optimal schedule length is the shortest time it takes to execute a task system on a given number of processors. The optimality criterion referred to in this research concerns producing optimal schedule lengths for individual jobs for the purpose

of program speed-up. Optimality of the schedule on a system-wide basis which is related to processor utilization and efficiency is not a focus of the current research. Bounds on optimal schedules will be discussed in this research along with the developed algorithms. Because of the NP-completeness of the scheduling problem, the goal of this research is to develop near-optimal scheduling algorithms that can schedule such task systems.

3.2 Task System Representations

The task system of a program (or a program segment) can be defined as $G(V, E)$, where G is a DAG. The set of vertices V of G represents the individual tasks in the graph. The set of arcs E represents the dependencies between pairs of vertices in the graph and thus imposes an order (in general, an irreflexive partial order) on the execution of the tasks. Any scheduling algorithm suitable for scheduling of such a task system must be able to take the execution constraints imposed on the graph by the arcs into consideration and schedule the tasks accordingly.

Once a job (a program) is divided into a number of tasks that can be assigned to different processors, depending on the functionality of each task, there are two possibilities:

i) all tasks are disjoint, or

ii) there are pairs of tasks that are dependent on one another.

In the first case, the tasks only compete for the system resources and no synchronization and/or sequencing of execution of individual tasks is necessary because there are no data or communication dependencies. The second case involves tasks that communicate with one another.

The problem of scheduling of task systems on a multiprocessor (consisting of a number of identical processors) can be divided coarsely into two categories which correspond to cases (i) and (ii) above, respectively.

a) Scheduling of $n > 1$ independent tasks on $p \geq 1$ processors. In this situation, the n tasks are independent of one another and they can be scheduled in any order (obviously, this is a special case of category (b) below).

b) Scheduling of $n > 1$ tasks on $p \geq 1$ processors where there may be dependencies among tasks. Unlike the previous category, the tasks in such a task system cannot be scheduled in any arbitrary order. The scheduling mechanism must take the existing precedence constraints into consideration in scheduling such task systems.

The main thrust of this research is developing algorithms that can produce schedules for each of the above categories of scheduling of a given task system. Since the general problem of producing optimal schedules has been proved to be NP-complete, the goal of this research is to develop algorithms that use heuristics to produce near-optimal schedules [Ullman 67] [Coffman et al., 78]. We refer to such a schedule as OPT and denote its length as T_{OPT}.

3.3 Task Systems with Independent Tasks

We start our discussions by concentrating on the first category presented in the introductory section, that is, the scheduling of n independent tasks on p processors. In such a case, given $G(V, E)$ which represents the task system, the edge set $E = \phi$ and $|V| = n$. Independent task sets are also known as *mutually non-interfering* task sets [Coffman and Denning 73]. Two tasks t_i and t_j are said to be non-interfering if *i)* t_i is a successor or predecessors of t_j, or *ii)* $R_{t_i} \cap R_{t_j} = R_{t_i} \cap D_{t_j} = R_{t_j} \cap D_{t_i} = \phi$, where R_{t_k} denotes the range, and D_{t_k} denotes the domain of task t_k for $1 \leq k \leq n$. A task system $\tau = \{t_1, t_2, ..., t_n\}$ is said to be mutually non-interfering if tasks t_i and t_j are non-interfering for all i and $j \in \{1 .. n\}$ and $i \neq j$. The issue of determinacy of tasks systems was discussed in Section 1.1. Independent task systems are determinate because the task set is entirely composed of mutually non-interfering tasks.

Since the tasks in an independent task system are mutually non-interfering, the case in which $n \leq p$ is trivial since task t_i can be scheduled on processor p_i for $1 \leq i \leq n$ and the $p-n$ other processors are idle or allocated to other task systems. The schedule length in this trivial case is the optimal schedule length represented as

$$T_p = \max_{1 \leq i \leq n} \{w(t_i)\}.$$

The cases where $n > p$ are less straightforward. Given p, the goal is to produce schedules that are as close to T_{OPT} as possible. The main goal is to reduce the schedule length as much as possible. Therefore, minimizing the number of processors needed is performed only if the resulting schedule does not increase in length. This is evidenced by the special case of $n \leq p$ discussed above.

Given p processors, we would like to balance the load on all processors such that, in the ideal case, all p processors finish execution of the assigned tasks at the same time. In other words, we would like to have T_p as close to T_s / p as possible. However, it is possible to have a task system with task weights assigned such that the execution time of the largest task on one processor exceeds the execution time of all $n-1$ other tasks on the $p-1$ other processors when $n \geq p$. Therefore, the following lower bound can be presented for the execution time of a task system,

$$LB = \max\{\lceil T_s / p \rceil, \max_{1 \leq i \leq n} \{w(t_i)\}\}$$

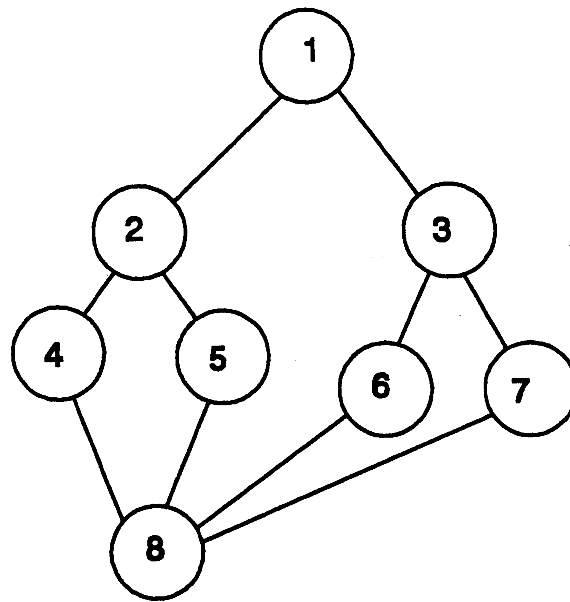
LB can be used as the ideal execution time or the schedule length on each of the p processors (occasionally, we refer to LB as T^*_{OPT}). Notice that in general $T_{OPT} \geq LB$; i.e., our goal is to produce schedules that are as close to LB as possible. The scheduling algorithm described in Section 4.3 is a near-optimal scheduling algorithm that uses LB as the ideal length of the schedule for each of the p processors and balances the load on each of the p processors around LB.

3.4 Task Systems with Dependent Tasks

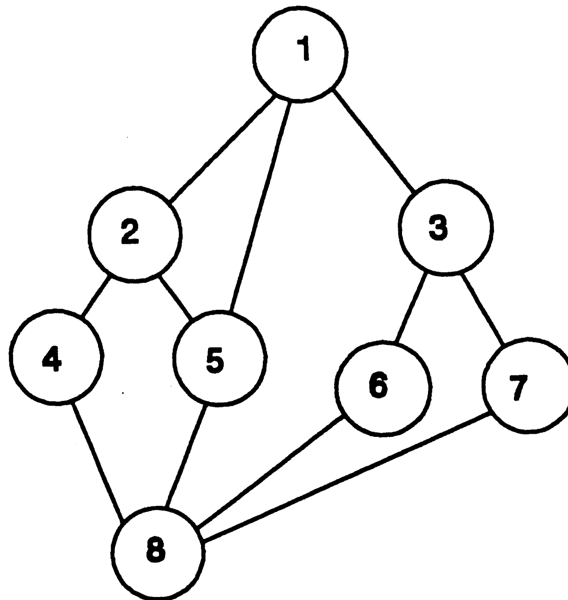
A task system consists of a set of tasks. We define a task set with n tasks, as $V = \{t_1, t_2, ..., t_n\}$. The symbol $<$ as a *relation*, represens a partial order on V. Therefore, a task system can be defined as $\tau = (V, <)$ where the partial order, $<$, represents the precedence relation. For example, $t_i < t_j$ signifies that task t_i must finish execution before task t_j can begin. In cases where $< = \phi$, it is said that the task system is entirely composed of mutually non-interfering tasks.

In the previous section, the graph representation of independent task systems was defined as $G(V, E)$ where $E = \phi$. Using the same notation, a dependent task system may be viewed as a graph in which $E \neq \phi$ or in other words, based on the above definition of a task system, the partial order set $< \neq \phi$.

In this dissertation, graph representation of task systems are assumed to be a DAG and that moreover, they are assumed to be precedence graphs. A precedence graphs is defined as follows. Define the arcs of a DAG by the ordered pair (t_i, t_j). The arc (t_i, t_j) is in the graph if and only if $t_i < t_j$ with the added condition that there exists no t_k such that $t_i < t_k < t_j$. Informally, the above definition guarantees that there are no redundant edges or paths in the DAG for it to be a precedence graph. Figures 1 (a) and 1 (b) demonstrate a typical precedence graphs and a general DAG. As can be seen, the addition of the arc (1, 5) to the precedence graph in Figure 1 (b) creates a redundant path from vertex 1 to vertex 5. As mentioned in the statement of the problem in Section 1.1, existence of an arc between a pair of tasks indicates inter-task dependencies which demands imposition of a partial ordering on the execution sequence of tasks. Because of the dependency constraints imposed on the task system graph through the existence of arcs, the scheduling mechanism must schedule the tasks in such a way that correct execution of the program is guaranteed.



(a) A precedence graph



(b) A directed acyclic graph (DAG)

Figure 1. A sample precedence graph and a sample DAG

3.5 Summary

This chapter presents a broad discussion of multiprocessor scheduling. Task systems are characterized in terms of the two classes of dependent task systems and independent task systems. Formal and informal definitions for each class are presented. Bounds on the length of schedules as related to the number of available processors are presented and discussed.

CHAPTER IV

SCHEDULING OF INDEPENDENT TASKS

4.1 Introduction

Given a task system (or a set of tasks) represented as a graph $G(V, E)$, where the number of tasks $n = |V|$, the tasks are said to be independent of one another if the edge set $E = \phi$. That is, there are no communication or data dependencies between any pairs of tasks. Bounds on the schedule length for scheduling of n tasks on p processors was discussed in Section 3.3. This chapter concentrates on the discussion of practical issues related to scheduling of independent tasks on multiprocessors and presents an algorithm that is designed for performing such a task.

Because of the absence of inter-task dependencies, scheduling of independent tasks can take place in any order. Additionally, no special considerations are necessary to determine the particular processor that a given task must be scheduled on in order to avoid or minimize the communication overhead. Such a concern is warranted in scheduling of tasks that are dependent on one another, particularly if the mapping of tasks to processors is attempted for distributed memory environments or machines.

The main objective in scheduling of independent tasks on a given number of processors p, is to balance the load on all processors such that, in the ideal case, all p processors finish processing of their workload at the same time. If such a goal is reached, and optimal schedule length is achieved. We refer to such an ideal schedule as τ_{opt} to indicate an optimal schedule for task system τ and denote its length as T_{OPT}.

The scheduling algorithm, *Variant-Load* algorithm, presented in Section 4.3 is designed for scheduling of independent tasks on a given number of processors p, with the objective of balancing the assigned workload to each of the p processors. Although this algorithms has been designed for scheduling of individual tasks constituting a single job, on multiprocessors, it also can be used effectively in a multiprogrammed multiprocessor environment for the purpose of load balancing.

Variant-Load algorithm treats each available processor as a *bin* with a certain *capacity* (workload capacity) that is filled (packed) with variable-sized processing times associated with the tasks. This algorithm is developed based on concepts from bin packing [Coffman et al. 78] [Johnson et al. 74]. Bin packing is a general scheme in which items of different sizes must be packed into k bins of capacity C each where the sum of the sizes of the n items being packed is in general less than or equal to $k*C$.

Bin packing is known as a combinatorial optimization problem [Johnson et al. 74]. Other similar problems include the traveling salesman problem, the least sum of squares problem, and the multiprocessor scheduling problem. A more formal statement of the bin packing problem can be stated as follows. Consider a list $L=(e_1, e_2, ..., e_n)$ of real numbers in the range $[0..1]$. The objective is to place the n numbers into the smallest possible number of bins L^* such that the sum of the numbers assigned to any of the bins would not exceed 1. It is possible to define other variations for the bin capacity. One possible variation is to consider bins with different capacities. Such a problem is referred to as the variable-size bin packing problem. Bin packing is a problem that has its origin in operations research and job-shop scheduling. It is considered to be a special case of two other problems known as the cutting stock problem and the assembly line balancing problem [Gilmore and Gomory 61] [Conway 67].

The cutting stock problem can be describes as a problem in which a number of inventory item of various lengths $L_1, L_2, ..., L_m$ are available for filling customer

orders. Each inventory item has a certain cost and therefore, the cost of filling orders is determined by which stock is cut. To fill an order of N_i pieces of various sizes l_i, for $1 \le i \le m$, inventory items must be selected for cutting. The problem to solve is which available inventory stock to cut in order to incur the lowest cost. This problem is also referred to as the trim problem. An instance of the cutting stock problem is demonstrated in Figure 2 where the numbers inside each box represents the size of the available inventory item or the ordered item.

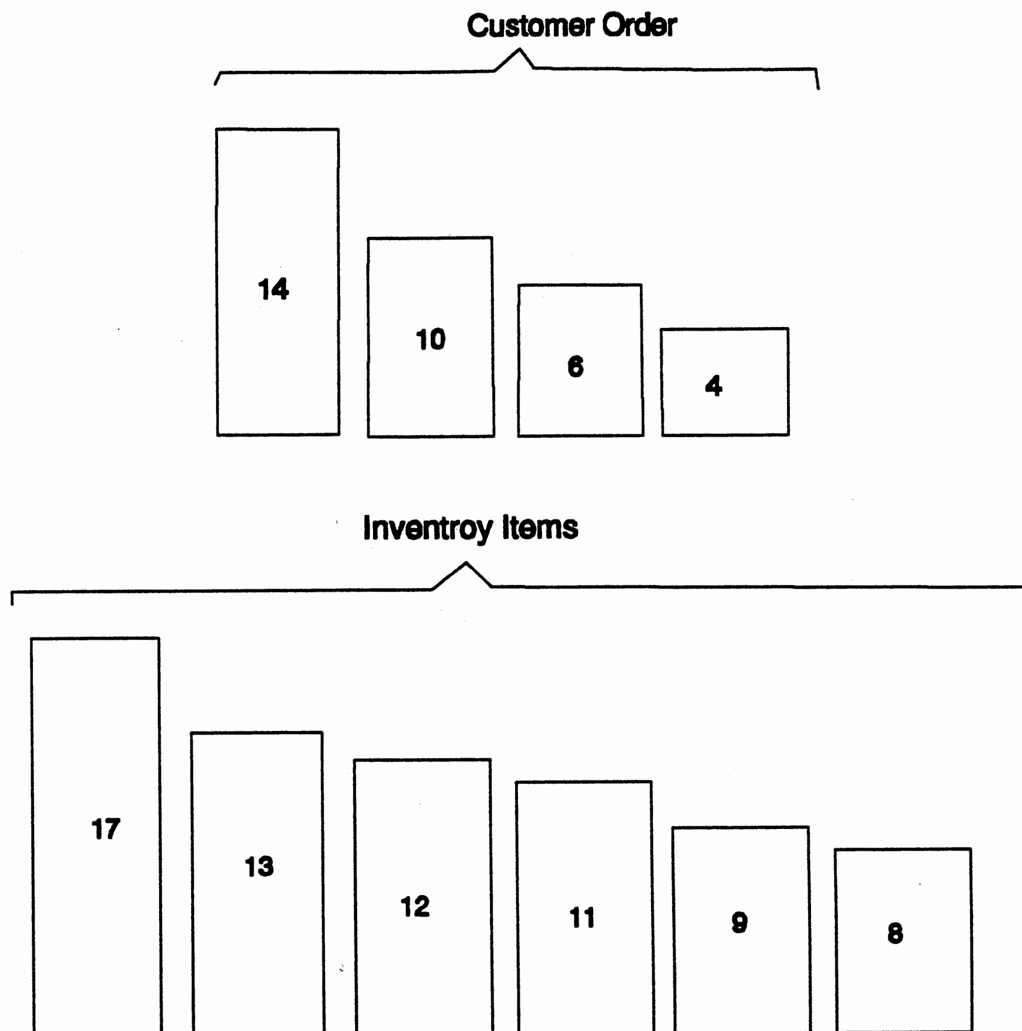


Figure 2. The Cutting Stock Problem

4.2 Survey of Related Work

Coffman, Garey, and Johnson [Coffman et al. 84] in an update survey of bin packing problem, present a mathematical model for the classical one-dimensional bin packing problem as "given a positive bin capacity C and a set or *list* of items $L=(P_1, P_2, ..., P_n)$, each P_i having a size $S(P_i)$ satisfying $0 \leq S(P_i) \leq C$. What is the smallest integer m such that there is a partition $\pi = B_1 \cup B_2 \cup \cdots \cup B_m$ satisfying

$$\sum_{P_i \in B_j} S(P_i) \leq C$$

for $1 \leq j \leq m$." This definition characterizes many different problems that arise in real-life situations in which a collection of given objects of different sizes are to be fit into well-defined regions without the possibility of overlaps. Johnson and his colleagues [Johnson et al. 74], in an earlier survey, present several examples of situations in which the need for bin packing algorithms arises. Such examples include *i) table formatting* in which "bins" are assumed to be computer words that hold such items as half-word integers, bit strings of certain sizes, and character strings, *ii) prepaging* where the bins are assumed to be pages or page fractions, and *iii) file allocation* problem where the need for placing variable-sized files on as few disc tracks as possible is felt.

Other variations of the classic one-dimensional bin packing problem are two dimensional bin packing problems in which objects are assumed to posses certain widths and heights, and three dimensional bin packing problems in which objects may be viewed as cubes with varying dimensions. Most computer science problems are best characterized as belonging to the class of one-dimensional bin packing. Brown [Brown 71] gives examples of applications for business and industry.

As stated in the introduction section of this chapter, the bin packing problem is known to be a combinatorial optimization problem and therefore, the general problem of finding

an optimal packing solution requires a lengthy combinatorial search and is NP-complete [Cook 71]. As a result, similar to other attempts at finding reasonable solutions for other NP-complete problems, one must resort to finding heuristic solutions for creating acceptable packing of items into bins. We refer to such heuristic algorithms as approximation algorithms. Johnson and his colleagues [Johnson et al. 74] present some worst case performance bounds on four approximation algorithms for the one-dimensional bin packing problem. The heuristics used in these algorithms are first-fit (FF), best-fit (BF), first-fit-decreasing (FFD), and best-fit-decreasing (BFD). In the latter two cases, FFD and BFD, items are expected to be sorted in non-increasing order according to their sizes. Due to the sorting requirement in the case of FFD and BFD, these approximation algorithms can only be used for off-line bin packing where a static solution is reached before its implementation. A bin packing algorithm is said to be off-line if permutation of list items are allowed before processing. In contrast, on-line bin packing algorithms are capable of coming up with a packing of items on a dynamic basis as the item sizes become available. Therefore, such on-line algorithms, by their very nature, must be very fast and use simple heuristics. An investigation of on-line bin packing algorithms has been the topic of a dissertation by Ramanan [Ramanan 84] in which a linear-time on-line algorithm is presented and analyzed. It may be argued that off-line solutions may yield better results. However, one must also consider the overhead involved in preprocessing of the items.

Johnson et al., in analyzing the worst case and average case behavior of the packing algorithms described in the above paragraph, conclude that FFD and BFD almost always behave better than the FF and BF heuristics. However, FFD and BFD are only reported as best off-line algorithms. These algorithms are reported to have a run-time complexity of $O(n \log n)$ and a performance ratio of 1.222... which is defined as the ratio of the performance of the given approximation algorithm to the performance of an optimal solution known for the same problem.

Another related work is one done by Coffman, Gary and Johnson [Coffman et al. 78] that develops a multiprocessor scheduling algorithm using the FFD heuristic, called *multifit.* Multifit is a non-preemptive scheduling algorithm for scheduling of n independent tasks on p identical processors. Given the number of available processors p, this algorithm operates by determining a bin capacity C, and attempts to schedule the given tasks on $\leq p$ processors. Coffman, et al. define a *p-processors performance ratio*, $R_p(A)$ for a given algorithm A as:

$$R_p(A) = sup \{ F_A [\tau, p] / \tau_p^* : \text{all task sets } \tau \}$$

where τ and p represent the given task system and the available number of processors, respectively. Algorithm A creates a partition of τ into p subsets. The partitions are denoted by $\rho_A [\tau, p]$. $F_A [\tau, p]$, in the above performance formula, denotes the schedule length or the finishing time of $\rho_A [\tau, p]$. τ_p^* is defined to be the optimal schedule length for a given task system τ on p processors. The goal is to find an efficient algorithm A such that the performance ratio, $R_p(A)$, is as close to 1 as possible. Graham [Graham 69] [Graham 76] and Sahni [Sahni 76] discuss such algorithms. However, the computational time associated with these solutions make their practicality prohibitive. One of the polynomial time solutions that seems to have a good performance is the Largest Processing Time [LPT] algorithm [Graham 69] [Graham 76] that reports a performance ratio of $R_p(LPT) = 4/3 - 1/3p$. Graham also reports in these studies that the performance of non-preemptive scheduling algorithms for scheduling of independent tasks is in general, never worse than twice the optimal solution in length, that is, $F_A [\tau, p] \leq 2 . \tau_p^*$.

Coffman et al. [Coffman et al. 78] report that their iterative algorithm, multifit, out performs LPT on the average and improves on the worst case performance ratio of LPT. Given a task system τ, composed of independent tasks, bin capacity C, and the number of processors p, the multifit algorithm devises a *packing* of tasks for each of the

processors with the objective of minimizing the number of processors. However, it is described later that since Coffman et al. determine the bin capacity with regard to p, it is never the case that the tasks are scheduled on less than p processors. Instead, whenever the workload has to spill over to more than p processors, the bin capacity is increased until a suitable capacity is reached for which the n tasks can be fit into the p processors. We demonstrate that the algorithm developed in this dissertation, the *Variant-Load* algorithm, achieves the same goal with less processing.

Multifit algorithm determines the bin capacity C by defining the lower bound and the upper bound values on C with regard to p. It then performs a binary search for finding a new value for C which minimizes the schedule length by performing a limited space search. Each new value for C is used for creating a packing of tasks on processors. If the new packing requires more than p processors, it is rejected and a new value for C that lies between the defined lower and upper bounds is determined. During the *trial* packings, new lower and/or upper bounds on C are defined until a packing of items into no more than p bins is achieved. Because of the expense involved in searching for a reasonable schedule, the multifit algorithm operates such that the number of trial packings in search of a suitable C can be specified by the user, where fewer iterations provide a less accurate capacity and therefore the algorithm may be unable to pack the items in less than p bins. In this work, the lower bound C_L, is defined as:

$$C_L\,[\tau, p] = \max\{\,(1/p)\ T_s(\tau),\ \ \max\{\,w\,(t_i)\,\}\,\}$$

where $T_s\,(\tau)$ is the sequential execution length of a task system τ, and $w\,(t_i)$ is the processing time of the largest tasks in τ. This quantity is known as McNaughton's lower bound [McNaughton 59] and is devised as an optimal schedule length for preemptive scheduling. The upper bound C_U, is defined as:

$$C_U\,[\tau, p] = \max\{\,(2/p)\ T_s(\tau),\ \ \max\{\,w\,(t_i))\,\}\,\}$$

where the quantity $(2/p)\ T_s(\tau)$ is based on Graham's findings that the worst case performance of non-preemptive scheduling algorithms for scheduling of independent tasks is never more than twice the optimal schedule length. Therefore, multifit algorithm searches for a C value in the range $C_L \leq C \leq C_U$.

One interesting finding of the experiments done by Coffman et al. (which is also demonstrated in [Johnson 73]) is non-monotonocity of the devised packings with respect to the bin capacity and the number of employed bins. This anomaly is demonstrated in the example in Figure 3 which appears in [Coffman et al. 78]. Given a task system τ = {44, 24, 24, 22, 21, 17, 8, 8, 6, 6}, if the bin capacity is set to $C = 60$, the number of bins required to devise a non-overlapping packing is three while if the bin capacity is increased to 61, then packing of the same set of tasks (using the FFD heuristic) requires

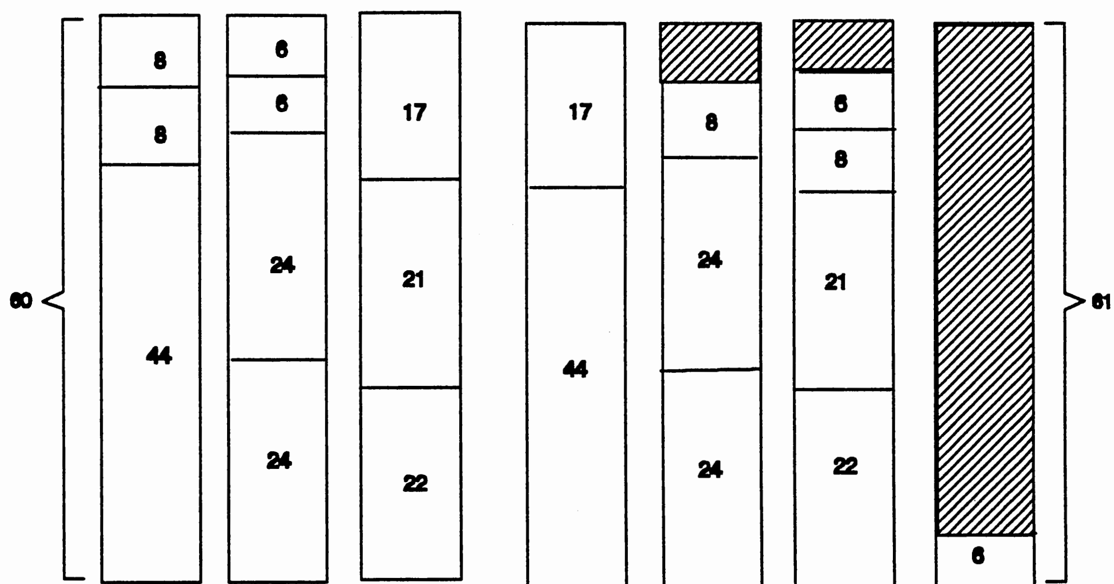


Figure 3. Non-monotonocity of First-Fit-Decreasing heuristic in packing of items into bins

four bins. Johnson [Johnson 73], in his dissertation work on bin packing algorithms, demonstrates the same anomaly with regard to First-Fit heuristic. Figure 4 demonstrates packing of tasks $\tau_1 = \{.55, .70, .55, .10, .45, .15, .30, .20\}$ and $\tau_2 = \{.55, .70, .55, .45, .15, .30, .20\}$ in which the sum of the values of all items $\tau_s = 3$ for both task systems. The bin capacity in Johnson's work is assumed to be 1. As seen in Figure 4 (which appears in [Johnson 73]), First-Fit heuristic comes up with an optimal solution for task system τ_1 while τ_2 is packed into four bins. Prior to the demonstration of non-monotonocity of FF and FFD heuristics, a more detailed study of scheduling anomalies was done by Graham [Graham 66] when discussing the *list scheduling* heuristics. A detailed discussion of Graham's findings was presented in chapter one.

The issue of monotonocity of heuristic bin packing algorithms is the subject of a dissertation work by Murgolo [Murgolo 85]. It is reported in this work that from among

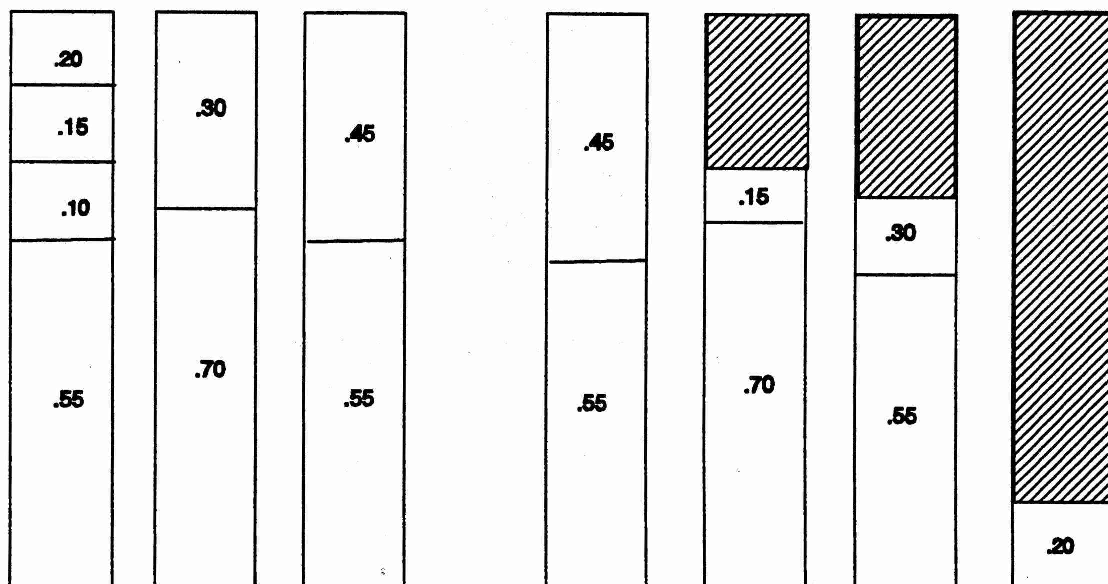


Figure 4. Non-monotonocity of First-Fit heuristic in packing of items into bins

the heuristics first-fit, best-fit, worst-fit, next-fit and their off-line variations, first-fit-decreasing, best-fit-decreasing, etc., only the next-fit heuristic and one of its variations next-fit-2, is reported to be monotonic. For such algorithms to be monotonic, it means that given a list L and its variation L', which differs from L by containing items of smaller sizes and therefore, a smaller total for the sum of its individual item sizes, a packing of L' must require the same or fewer number of bins as L. Otherwise, the algorithm is not monotonic.

Other combinatorial allocation problems related to the bin packing problem are the knapsack problem and the minimum sum of squares problem. Knapsack problem is analogous to multidimensional bin packing problem in the sense that the value of an item in the knapsack problem is determined as a ratio of different value factors attributed to the objects being packed into the knapsack. Different theoretical models for both of these problems are presented in the dissertation research by Neilsen [Neilsen 85].

Another related work designed for scheduling of independent tasks on a given number of processors is *Divide & Fold* (D & F) algorithm by Polychronopoulos [Polychronopoulos 86]. The main objective of D & F, same as other similar algorithms, is to devise a schedule with the shortest possible schedule length, given a set of independent tasks and a number of processors. D & F algorithm operates under two phases. It starts with a list that is sorted in non-ascending order based on the task processing times. During phase I, it repeatedly divides the sorted list in half and folds the two halves into one list. Assuming that the list starts with n partitions of one element each, after the first division and folding, the list consists of $n/2$ partitions, each with two elements. The dividing and foldings continue until the number of partitions created is equal to the number of processors and each partition contains n/p tasks. Figure 5 demonstrates the first phase of the D & F algorithm for the task system $\tau = \{45,\ 41, 32,$

28, 22, 20, 19, 17, 8, 5, 4, 1} to be scheduled on three processors.

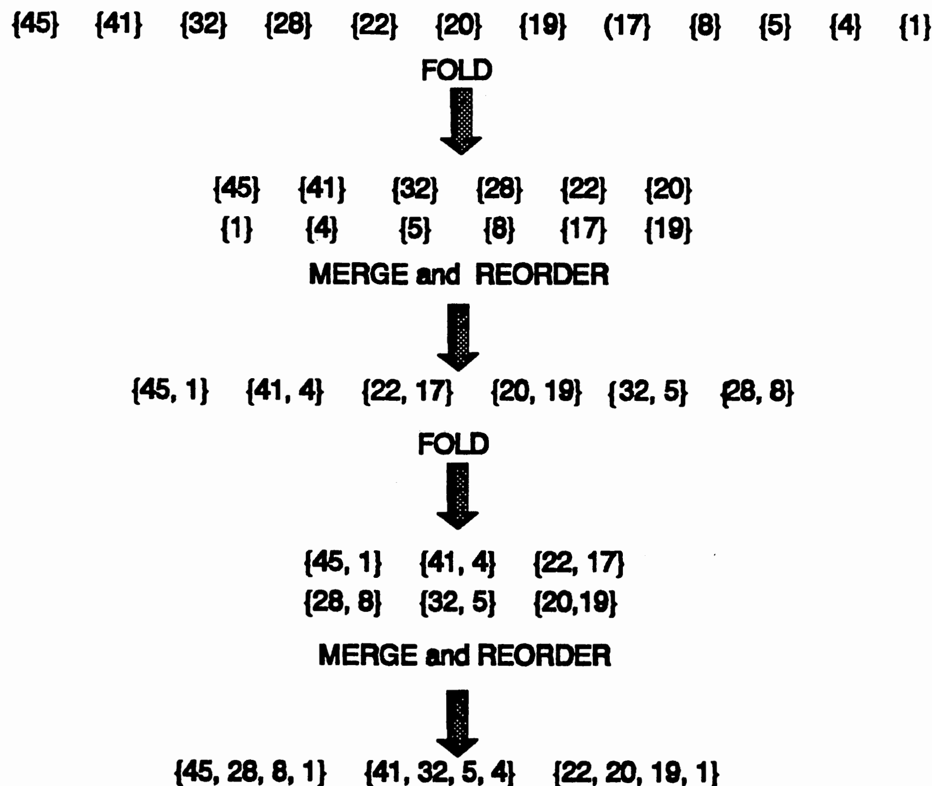


Figure 5. Phase I reordering of D & F

The second phase of the D & F algorithm is the balancing phase that performs three different tests in order to identify tasks from pairs of processors whose swapping will further balance the workload assigned to each of the processors. During the second phase, the workload assigned to pairs of processors $(i, p-i+1)$ for $i = 1, 2, ..., \lceil p/2 \rceil$ is examined and if any tasks are found that can be swapped with tasks assigned to the other processors or transferred to the other processors in order to balance the load, such exchanges or transfers will take place. For example, in Figure 5, the workload assigned

to the three processors after phase I, is 82, 84, 78. One obvious transfer of assignment upon examination of tasks in the first and the third partition would be to move the task with the processing time of 1 in the first partition, to the third partition that has a lighter workload. The decision about which pairs of tasks to exchange or which task to reassign is done based on the outcome of three different tests that are performed during the second phase.

The complexity of Divide & Fold algorithm is reported as follows. Given an initially sorted list, phase I of this algorithm requires $O(log_2 (n/p))$ steps to complete. Each step consists of dividing, merging, and reordering of the newly created partitions. Phase II of D & F consists of comparison of the workloads assigned to pairs of processors. This phase of the algorithm is considered to be the bottleneck because of its $O(n^2/p^2)$ complexity. Polychronopoulos reports the same complexity for the multifit algorithm developed by Coffman et al. [Coffman et al. 78].

4.3 A Near-Optimal Scheduling Algorithm for Scheduling of Independent Tasks

The scheduling algorithm presented in this section is based on concepts from bin packing [Coffman et al. 78] [Johnson et al. 74]. As discussed in the introduction section, bin packing is a general scheme in which n items of different sizes must be packed into k bins of capacity C each where the sum of the sizes of the n items is in general less than or equal to $k*C$.

In Variant-Load algorithm discussed below, the number of bins k is set to the number of available processors p. The initial capacity C of all the bins is set to LB, as defined in Section 3.3, before the process of assigning tasks to bins is initiated. The tasks in the task system τ are arranged in non-ascending order based on their sizes such that $w(t_1) \geq w(t_2) \geq \cdots \geq w(t_n)$ where $w(t_i)$ represents the execution time (weight) of

tasks for $1 \leq i \leq n$. The tasks in each bin, b_j, $1 \leq j \leq p$, correspond to the set of independent tasks assigned to each of the p processors. After partitioning of the tasks into p sets, the tasks assigned to bin b_j can be scheduled on processor p_j, $1 \leq j \leq p$. $w(b_j)$ will be used in the following algorithm to denote the sum of the weights (execution times) of all the tasks assigned to bin b_j.

The Variant-Load algorithm, presented in this section, performs the task of assigning the first P tasks of the given tasks set, one task per processor, to the p available processors through the first two steps of the algorithm. If $n \leq p$ (the trivial case), then the assignment of tasks to processors is completed and the algorithm halts. The other non-trivial case where $n \geq p$ is performed by going through step 4, or if required through steps 4 and 5 of the algorithm.

After the initial assignment of the first p tasks to the p processors, the algorithm starts assigning the remaining $n-p$ tasks as follows. Tasks $p+1$ to n are assigned to the lowest indexed bin (processor) provided that the assigned workload, $w(b_j)$, for the lowest indexed bin, b_j for $1 \leq j \leq p$, does not exceed the defined bin capacity, LB. If this condition is satisfied, then task t_i, $p+1 \leq i \leq n$, is added to bin b_j and the workload assigned to bin b_j so far, is incremented by $w(t_i)$, the execution time for task t_i. Step 4 will be repeated until $i = n$, that is, all tasks are packed, or until the assignment of a new workload to a bin exceeds the bin capacity LB. The latter condition forces the algorithm to step 5, with the understanding that the current bin could not be packed with a new task.

During step 5 of the algorithm, the simple initial "if" test determines if the current bin being packed has not been in fact the highest-indexed bin. If so, the algorithm simply moves to the next lowest-indexed bin and transfers the control back to step 4, for new attempts at packing the current task into the next bin in line. If $j = p$, that is attempt to

---

Input: A task system τ, consisting of independent tasks, where the tasks are arranged in non-ascending order based on their execution weights, the number of tasks n in τ, the number of available processors p, and the initial bin capacity LB.

Output: A partitioning of τ into p sets.

Method:

1. $b_k \leftarrow \{t_k\},\ w(b_k) \leftarrow w(t_k),\ 1 \le k \le p.$
2. if $n \le p$ then HALT.
3. $i \leftarrow p+1,\ j \leftarrow 1.$
4. if $w(b_j)+w(t_i) \le LB$

   then

   $b_j \leftarrow b_j \cup \{t_i\},\ w(b_j) \leftarrow w(b_j)+w(t_i),$

   $i \leftarrow i+1,$

   if $i \le n$ then go to Step 4, otherwise HALT.
5. if $j = p$

   then

   find k such that $w(b_k) \le w(b_l),\ 1 \le l \le p,$

   $b_k \leftarrow b_k \cup \{ t_i \},\ w(b_k) \leftarrow w(b_k)+w(t_i),$

   if $i < n$

      then

      $i \leftarrow i+1,\ j \leftarrow 1,$ go to Step 4,

      otherwise HALT.

   otherwise $j \leftarrow j+1,$ go to Step 4.

Figure 6. The Variant-Load Algorithm

pack a task in the highest-indexed bin has failed, then step 5 of the Variant-Load algorithm recognizes that packing of the current task without exceeding the bin capacity is (probably) not possible. Therefore, it attempts to pack the current task t_i into bin b_k, $1 \le k \le p$ that has the lightest workload $w(b_l)$ for $1 \le l \le p$. Now, task t_i has been added to bin b_k. The algorithm halts when the number of iterations, $i = n$. Otherwise, the next task, t_{i+1}, becomes the candidate for packing. After each failed attempt at packing a task, due to exceeding of the initial bin capacity, packing of subsequent tasks take place by examining the next lowest-indexed bin. Except for the initial assignment of the p largest tasks which is done using the scheme Largest-Processing-Time-First (LPTF), the scheduling of the remaining $n-p$ tasks is done using the First-Fit-Decreasing (FFD) scheme [Coffman, et al. 78] [Johnson, et al. 74]. The schedule in Figure 7 shows the task assignments produced by Variant-Load algorithm for $p=3$ and the task system $\tau = \{t_1, t_2, ..., t_{12}\}$ in which the processing times for the tasks are (45, 41, 32, 28, 22, 20, 19, 17, 8, 5, 4, 1). Processing times used for scheduling of the tasks typically are based on estimates provided by the compiler or the actual execution times obtained by sequential execution of the tasks. The lower bound on the schedule length for this particular task system is $LB = \max \{81, 45\}$. The schedule in Figure 7 is indeed optimal. If the packing scheme used in Variant-Load algorithm is changed either to a strict FFD or LPTF, the schedule increases in length from 81 to 82.

The rationale for using the LPTF scheduling for the first p tasks followed by a switch to FFD instead of using the LPTF or FFD schemes strictly, is that by placing the first p largest tasks in separate bins, the packing of the remaining (smaller) $n-p$ tasks will intuitively yield a more normalized distribution.

The packing scheme presented in Variant-Load algorithm is using the rationale of the worst-fit placement strategy implicitly in the following sense. By choosing not to place tasks t_i and t_{i+1}, $1 \le i \le p$ in the same bin, even if $w(t_i) + w(t_{i+1}) \le LB$, this

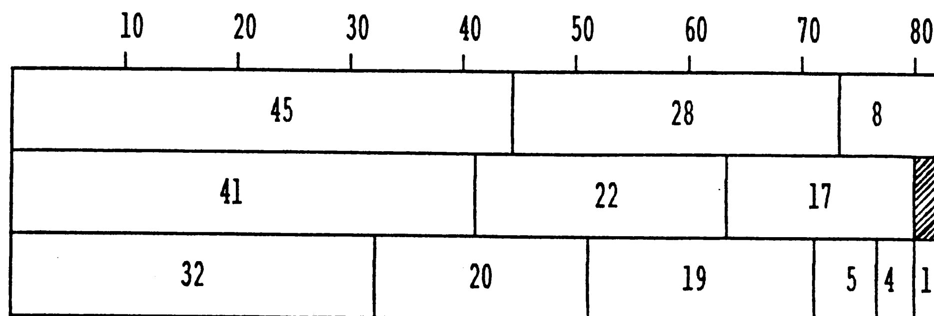


Figure 7. A schedule for task system τ on three processors

algorithm is in essence trying to maintain the largest possible remainder capacity in all the bins after the placement of the initial p tasks, with the expectation that it can still accommodate other incoming tasks with large weights.

4.4 Performance Evaluation

In order to evaluate the performance of the Variant-Load algorithm empirically, we compare the performance of this algorithm with several other comparable and well-known algorithms. The algorithms used in our simulation experiments are Longest Processing Time (LPT) [Johnson et al. 74], Multifit [Coffman et al. 78], and Divide & Fold (D & F) [Polychronopoulos 86]. All of these algorithms are suitable for scheduling of independent serial tasks. The performance of the three algorithms mentioned above, has been reported in the literature [Coffman et al. 78] [Polychronopoulos 86] in increasing order of performance as follows: LPT, Multifit, and D & F.

4.4.1 Design Methodology

The measure of performance in our empirical evaluation is the number of optimal schedules produced in a series of simulation experiments as well as the performance ratio of the measured algorithms. The variables used in these experiments are the processing

time of tasks, the number of available processors, and the number of tasks in the task system. More specifically, two types of experiments were conducted. The input ranges used in this experiment were the same as the ones used in the cited studies (namely, [Coffman et al. 78] and [Polychronopoulos 86]) in order to establish a baseline for further evaluation.

Two different simulation experiments were performed. In the first experiment, the number of tasks were treated as the independent variable while the processing times and the number of processors were treated as dependent variables. That is, the number of tasks was kept constant at 128 while the processing times ranged in value from 1 to 100, and the number of processors used was in the range 2 to 21. This type of experiment involved a total of twenty runs for each of the algorithms. In the second experiment, the number of processors was the independent variable while the processing times and the number of tasks were treated as dependent variables. More specifically, the number of processors was kept constant at $p = 10$ while the number of tasks in each task set varied as 20, 30, 40, ..., 210, and the processing times were selected randomly in the range 1 to 100. The processing time of tasks were produced using the normal distribution for both experiments. However, the same experiments were repeated with exponentially distributed processing times with the finding that the results were consistent with those of the normal distribution. The execution times of tasks were produced using the normal distribution for both experiments.

4.4.2 Simulation Results

The performance of the studied algorithms was measured in terms of two criteria, the number of optimal solutions achieved and the overall performance ratio of each algorithm. As described in Section 1.6, the performance ratio $R_p(A)$, is defined as

$$R_p(A) = T_A / T^*_{OPT}$$

where T_A is the performance of an approximation algorithm A and T^*_{OPT} is the performance measure for the approximation of an optimal solution known for the same problem. To compare the evaluated algorithms, we concentrated on the average performance ratio of these algorithms over all twenty runs. We denote the average performance ratio by $\overline{R_p(A)}$ which is the ratio of $\overline{T_A}$ to $\overline{T^*_{OPT}}$ representing the average performance for the approximation algorithm and the optimal solution, respectively. It should be stated that since it is difficult to determine exact values for T_{OPT} for arbitrarily large task sets, T^*_{OPT} was used as the preemptive measure and defined as

$$T^*_{OPT} = \max\{\lceil T_s/p \rceil, \max_{1 \le i \le n} \{w(t_i)\}\}$$

where T_s denotes the sequential execution time of the task set and $w(t_i)$ for $1 \le i \le n$, represents the individual processing time of tasks. Since the scheduling algorithms evaluated in this dissertation are non-preemptive, it can be asserted that in general, it is possible that $T_A > T^*_{OPT}$ and nontheless, still be an optimal solution under non-preemption.

TABLE I
RESULTS OF EXPERIMENT I

ALGORITHM	NUMBER OF OPTIMAL SCHEDULES	PERFORMANCE RATIO
Variant-Load	18	1.0009
LPT	3	1.0027
D&F	16	1.0006
Multifit	4	1.18

TABLE II
RESULTS OF EXPERIMENT II

ALGORITHM	NUMBER OF OPTIMAL SCHEDULES	PERFORMANCE RATIO
Variant-Load	17	1.0031
LPT	0	1.0061
D&F	15	1.014
Multifit	0	1.031

The results of experiments I and II are shown in TABLES I and II. As can be seen, the performance of the Variant-Load algorithm is slightly better than D & F algorithm in experiment II and is very close to D & F in experiment I. Even though these results are based on limited experimentation, we may conclude that, in terms of the quality of the produced schedules, our algorithm is at least as good as D & F which has been shown to improve over other best known algorithms (Multifit and LPT) in some respects. However, our algorithm possesses a major advantage over D & F and Multifit with respect to its run-time behavior.

4.4.3 Complexity Analysis

D & F devises schedules using a three-phased scheme. The first phase of D & F algorithm has been reported to require $O(\log_2 (n/p))$ steps. Each step consists of dividing the list in half, merging of the two halves (folding), and reordering the newly created partitions according to the sum of the sizes of the tasks in each partion. The second phase of D & F has been reported to have a complexity of $O(n^2/p^2)$. The third phase of the algorithm, which performs further optimization through rather expensive

steps, has not been implemented by its developer when testing the performance of this algorithm.

The Variant-Load algorithm has $O(n)$ complexity in cases where $n \leq p$. For $n \geq p$, we evaluate the complexity of our algorithm as follows. Let us make the assumption that after the initial assignment of the first p tasks (one per processor), the remaining $n-p$ other tasks do not fit in any of the bins without exceeding the bin capacity. This is a pessimistic worst case assumption since the initial bin capacity has been defined in terms of the processing times of the tasks and therefore, one may expect that the initial bin capacity will not be broken at this point. Nonetheless, this assumption is made to come up with a worst case behavior. Let us make another worst case assumption that all bins are examined before it is found out that a task does not fit into any bin for all the remaining $n-p$ tasks and thus $(n-p)p$ iterations are used in step 4. Again, this is a pessimistic assumption because in step 4 of the algorithm, the search does not start from the lowest indexed bin every time. Assuming a linear search, finding the least full bin (step 5 of the algorithm) involves p comparisons each time. Therefore, a pessimistic worst case for the run time behavior of the Variant-Load algorithm is $(n-p)p^2$. A more realistic behavior is $(n-p)/2.p^2$ where $(n-p)/2$ of the tasks on the average may require examining of all p bins (in step 4 of the algorithm) before the search for finding a bin (without exceeding the initial bin capacity) may fail. A general worst case bound for the Variant-Load algorithm can be specified as $O(np^2)$. Recall from Section 1.2 that from among polynomial time approximation solutions for solving the scheduling problem, those with relatively slower growth rates (as the input size increases) are rated to be superior to others. The Variant-Load algorithm possesses such a characteristic.

4.5 Summary

Chapter 4 describes the general problem of scheduling of independent serial tasks and their schedule bounds. A review of related work and their results are presented which

includes early theoretical work on the scheduling problem in other disciplines such as operations research and its subfield job-shop scheduling, and management science. This review also includes the most recent work done in multiprocessor scheduling.

A near-optimal scheduling algorithm, Variant-Load algorithm, developed in the current research is presented and discussed. An empirical evaluation of the developed algorithm in comparison to the best known algorithms in the literature is presented. Although the simulation studies performed are limited, it is shown that our algorithm is at least as good as one of the best-known algorithms (D & F) in terms of the optimality of the resulting schedules and is superior to D & F in terms of its run-time complexity.

CHAPTER V

SCHEDULING OF DEPENDENT TASKS

5.1 Introduction

This chapter concentrates on the problem of scheduling of n tasks on p processors in which the tasks exhibit inter-task dependencies. Using the notation defined in Section 3.4, such task systems are defined as $\tau = (V, <)$ such that $< \neq \phi$, where the relation $<$ is a partial order. Task systems are represented as graphs. Furthermore, they are assumed to be precedence graphs. Because of the dependency constraints imposed on the task system graphs through the existence of arcs, the scheduling mechanism must schedule the tasks in such a way that correct execution sequence of the program is guaranteed. This property is the determinacy property from Section 1.1.

Two different approaches are used in addressing this problem. In the first approach, we demonstrate a method and present an algorithm that breaks the inter-task dependencies by partitioning the task system into sets of independent tasks. Each such partition or set is referred to as a *layer*. The tasks within each partition or layer are ordered and processed based on a partial order. However, the order in which the resulting layers are processed must obey a total order in order to satisfy the determinacy property. Two algorithms have been developed that first use the above approach and eliminate inter-task dependencies by partitioning the task system into layers of independent tasks. These algorithms then schedule the tasks in the resulting independent layers according to certain criteria. The difference between these algorithms and their suitability for different environments is discussed in this chapter.

The second approach developed is one in which sets of tasks are identified and treated as independent threads. Each thread is scheduled on a single processor. In this approach, an acyclic control flow graph of a program is used as a basis for detection of parallelism. Tasks in this model are treated as sequential blocks of code with no branching into or out of them except at the beginning and/or at the end, and with clearly defined input and output parameters. Parallelism in such a graph is manifested by the number of independent paths or threads of execution. The total number of independent paths in a graph is the *nullity* of the graph which is the number $e-n+2$, where e represents the number of edges and n represents the number of nodes in a graph [Berge 73] [Temperly 81]. Therefore, we define the nullity of the graph as the minimum number of processors needed for maximum execution speed. Further details on the approaches used in scheduling of dependent task systems will be presented in appropriate sections in this chapter.

5.2 Task System Graphs and Schedule Bounds

We represent task systems as directed acyclic graphs with no redundant paths (i.e., precedence graphs). Task graph representations that are not acyclic, can be convert to DAGs by merging the strongly connected components by using the graph's boolean connectivity matrix. DAGs with redundant paths can be converted to precedence graphs by using redundant path removal algorithms. Since this research is not concerned with program partitioning, it is assumed that a task system has been created using an optimal or near-optimal partitioning scheme and is therefore suitable for parallel processing. The algorithms presented in this chapter have been designed with precedence graphs in mind that start with a single root node (source) and end in a single terminal node (sink). In the absence of a single source and/or a single sink, a dummy node with a processing time of zero can be added to the graph. Thus, the algorithm can be applied to arbitrary precedence graph, including the tree graph considered in Hu's algorithm. A typical

precedence graph considered in this research is shown in Figure 8.

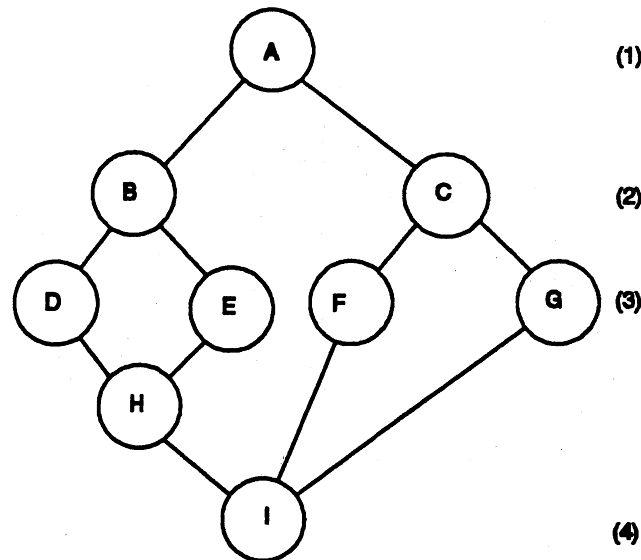


Figure 8. A task system precedence graph

As before, task systems will be represented as $G(V, E)$ where the set $E \neq \phi$ in the case of dependent task systems. The set of vertices represent the individual tasks. Associated with each vertex is a non-negative integer which represents the processing time of a task. Processing times are assumed to be compiler generated estimates or actual processing times obtained by running the tasks on actual processors. It is not possible to calculate the exact schedule lengths for non-preemptive scheduling schemes unless restrictive and sometimes unrealistic constraints are placed on the task and the machine characteristics. Calculation of the optimal schedule length is possible in preemptive schemes. Since the scheduling algorithm developed in this research uses a non-preemptive scheme, only loose bounds on the schedule length can be determined. Considering precedence graphs with a single source and a single sink (e.g., Figure 8), it is obvious that the source and the sink nodes cannot be executed in parallel with any other node. Therefore, that portion of

the task system which might be amenable to parallelization includes those tasks that are between the source and the sink. That is, parallel processable time at most is $t_s - (t_{source} + t_{sink})$. Therefore, a lower bound for the execution of the parallel processable portion of such a task system can be defines as

$$L1 = (t_s - (t_{source} + t_{sink}))/p$$

However, considering a precedence graph such as the one in Figure 9, it can be observed that $L1 = 11$, which is not achievable under a non-preemptive scheduling scheme. Thus, a second lower bound which is based on the critical path length, T_c, of a task system can be defined as follows

$$L2 = T_c - (t_{source} + t_{sink}).$$

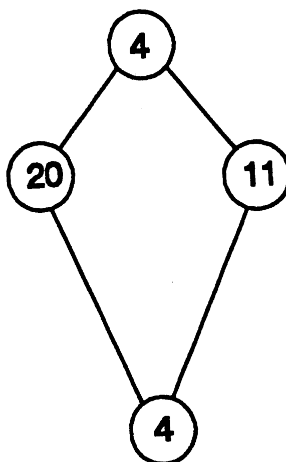


Figure 9. A simple precedence graph

Using these two bounds, a loose lower bound, LB, for the scheduling of the entire task system can be defined as:

$$LB = \max\{L1, L2\} + t_{source} + t_{sink}$$

Thus the schedule length for the graph of Figure 9 is 28 and not 19. In the case of an unlimited number of processors, or in cases where the number of available processors is greater than or equal to the width of the graph, the schedule length would be equal to the critical path length. In that case, $T_p = T_c = T_{OPT}$ and an optimal schedule length would be achieved.

5.3 Task System Partitioning

In order to eliminate inter-task dependencies in dependent task systems, many scheduling algorithms discussed in the literature make the assumption that the graph representation of the program has been partitioned into layers of independent tasks. There are very few algorithms (in fact only one) found in the literature that show how a task system graph is actually divided into layers. One such algorithm is MBFS [Polychronopoulos 86] which will be described in more detail later in this chapter.

Partitioning of task system graphs can be done using two different approaches. In what we refer to as the *earliest schedule partitioning* (ESP), the task system graph is processed from top to bottom. In this approach, any initial tasks that have no predecessors are placed in a layer, l_1. At this point, the arcs that lead to the immediate successors of the tasks in l_1 are erased. The same approach is repeated for the next layer in which all tasks with an empty predecessor set are now placed in layer l_2, followed by removal of outgoing arcs for the tasks in l_2. Repeated applications of this scheme will result in a set of independent layers $L = \{l_1, l_2, ..., l_k\}$ where k represents the number of independent layers in a graph. For example, in the case of precedence graphs described in this research where task graphs are assumed to be single-entry and single-exit graphs, the source node is assumed to be in layer l_1 and the sink node is placed in layer l_k. Most other work, following Hu's convention [Hu 61], assign labels from bottom up where the deepest level in the graph is assigned label l_1. We believe it is

more logical and practical to assign label l_1 to the first cut of the graph when the graph is processed from top down. We reserve the conventional labeling numbers for the next approach in which a graph is processed from bottom up.

If the partitioning scheme described above is applied to a graph by processing the task system graph starting at the sink node and working up to the source, then the resulting partitions are referred to as the *latest schedule partitioning* (LSP). Again, such partitioning will result in a set of layers $L = \{l_1, l_2, ..., l_k\}$ where l_k represents the first cut of the graph and l_1 represents a set containing the terminal node(s). In earliest schedule partitioning, presence of a task in a particular layer signifies the earliest time a task can be scheduled after all of its predecessors in lower-indexed layers have finished execution. Latest schedule partitioning, on the other hand, represents the latest time that a task must be scheduled without incurring a penalty, of course after all the tasks in the higher-indexed layers have finished execution. Figure 10 shows the result of earliest schedule partitioning applied to the task system precedence graph in figure 13. Figure 11 demonstrates latest schedule partition for the same task system graph. The algorithm presented in the next section, ESP algorithm, is designed for partitioning of task system graphs into sets of independent tasks using the earliest schedule partitioning approach. However, a bottom up application of the ESP algorithm yields the latest schedule partition.

5.4 An Algorithm for Task System Partitioning

The algorithm presented in this section partitions a task system graph into k layers denoted by l_i, $1 < i \leq k$, such that all tasks t_i, $1 \leq i < |V|$, in layer l_i are independent of one another. The ESP algorithm uses the concepts from *topological sorting* [Skvarcius and Robinson 86]. A topological sorting algorithm is an algorithm which imposes a total order on the nodes of a DAG by constructing a linearized list of the nodes in a DAG through relabeling its nodes. This algorithm, in its original form (i.e., before any

modifications), can be used as a scheduling algorithm for scheduling of the tasks in a task system graph on a single processor.

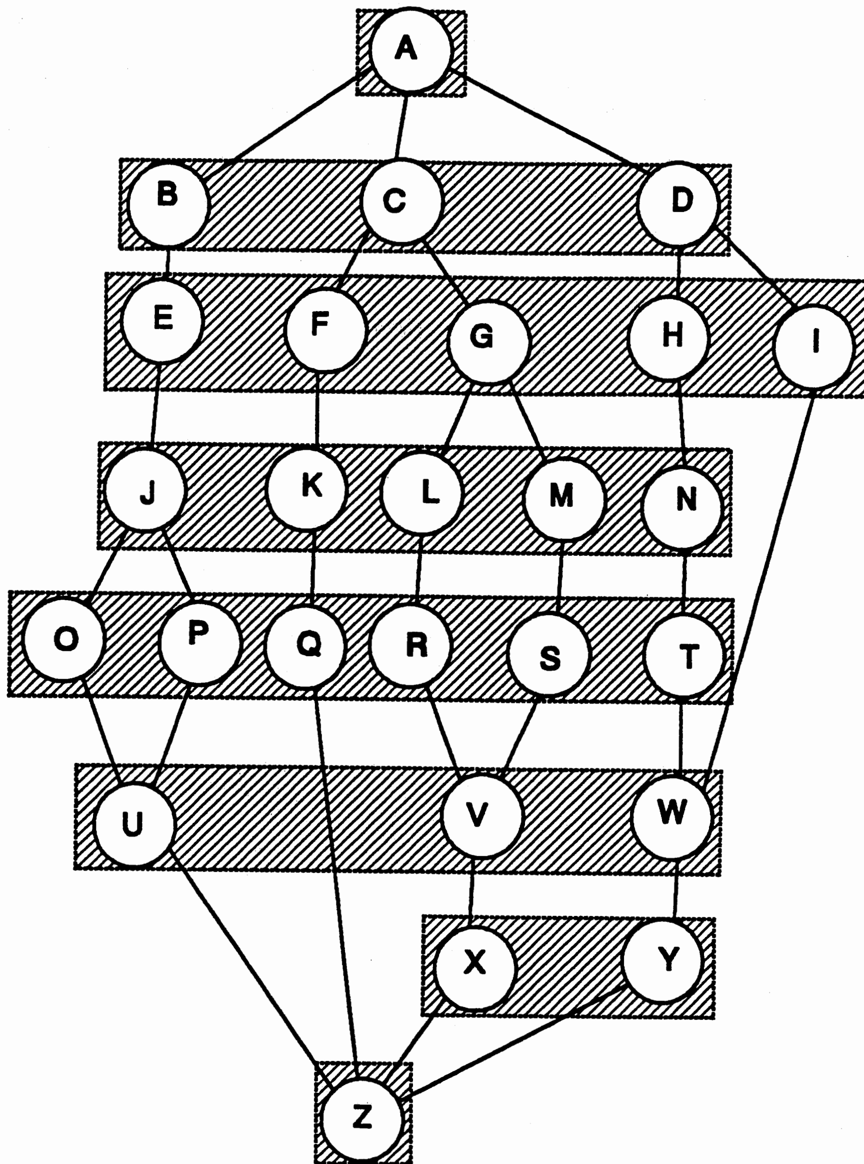


Figure 10. Earliest Schedule Partition (ESP) for the task graph in Figure 13

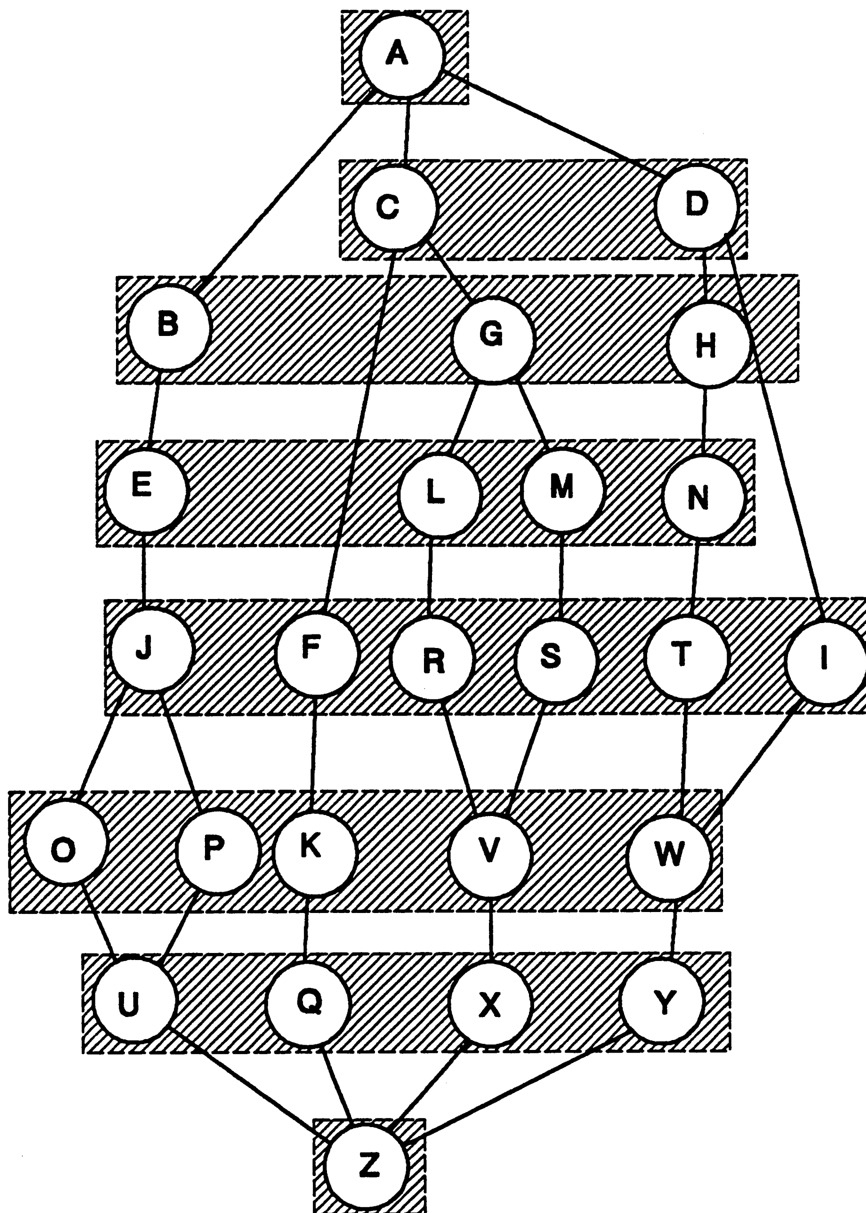


Figure 11. Latest Schedule Partition (LSP) for the task graph in Figure 13

In a topological sort, whenever two or more nodes are identified as having no predecessors, one is chosen arbitrarily for deletion from the DAG for relabeling. To identify as many independent tasks as possible in a task system, a modified version of the topological sorting algorithm can be used such that any number of nodes, which are identified as having no predecessors, are added to a list l_i at iteration i. The output of this algorithm consists of k such lists with k as the number of layers containing the independent task sets. Each list l_i is identified as a layer. The identified tasks in each layer can be scheduled in parallel, each on a separate processor, if available.

Polychronopoulos [Polychronopoulos 86] has developed an algorithm (MBFS) based on a modified breadth-first-search traversal in order to identify layers of independent tasks in a graph. MBFS relabels a vertex with higher label numbers until it is assigned its appropriate layer number. MBFS has an $O(n^3)$ complexity where the main loop is executed exactly n times. The dominant computation inside the main loop is the removal of the edges incident to a node and has a complexity $O(n^2)$. The main loop in our algorithm is executed exactly k times where k is the number of layers in a task system graph. Unlike MBFS which identifies the tasks that belong in the same layer one at a time, ESP algorithm identifies such tasks during the same iteration of the outer loop. ESP has a worst case complexity of $O(n^2)$. The worst case behavior occurs when there is no concurrency in the precedence graph (i.e., $k = n$). Under such a condition, the main loop is executed n times. As for the two nested for loops, the outer loop is executed exactly once because each layer consists of one task only. The inner loop, instead, is executed a maximum of n times for each of the n iterations of the while loop. The actual comparisons that take place in the inner for loop are $(n-1, n-2, ..., 2, 1)$. Therefore, a more exact value representing the worst case behavior of ESP algorithm is $n(n-1)/2$. The single for loop in the ESP algorithm is implemented as a list of sets, and therefore testing the loop condition requires a maximum of n comparisons.

Input: $G(V, E)$, where G is a DAG, and the tasks are each already assigned a label.

Output: Sets $l_1, l_2, ..., l_k$ of independent tasks where k is the number of independent task sets identified in G.

Method: 1. Compute PRED(v_i), $\forall\ v_i \in V,\ 1 \le i \le |V|$.

2. $V' \leftarrow V$; $k \leftarrow 0$.

3. while $V' \neq \phi$

    do

    $k \leftarrow k+1$; $l_k \leftarrow \phi$

    for all $v_i \in V'$ such that PRED(v_i) = ϕ do

        $l_k \leftarrow l_k \cup \{v_i\}$; $V' \leftarrow V' - \{v_i\}$

    endfor

    for all $v_i \in l_k$ do

        for all $u \in V'$ such that $v_i \in$ PRED(u) do

            PRED(u) $\leftarrow$ PRED(u) - $\{v_i\}$

        endfor

    endfor

endwhile

Figure 12. The ESP Algorithm

The task system in Figure 13 corresponds to an arbitrary program. The numbers placed by each node represent the execution times, $w(t_i)$, of each task t_i. One possible output of the topological sorting algorithm could be the total order A through Z. Applying ESP algorithm to the same task graph produces as its output eight sets {A}, {B, C, D}, {E, F, G, H, I}, {J, K, L, M, N}, {O, P, Q, R, S, T}, {U, V, W}, {X, Y}, and {Z}, identified as layers l_1 through l_8, respectively.

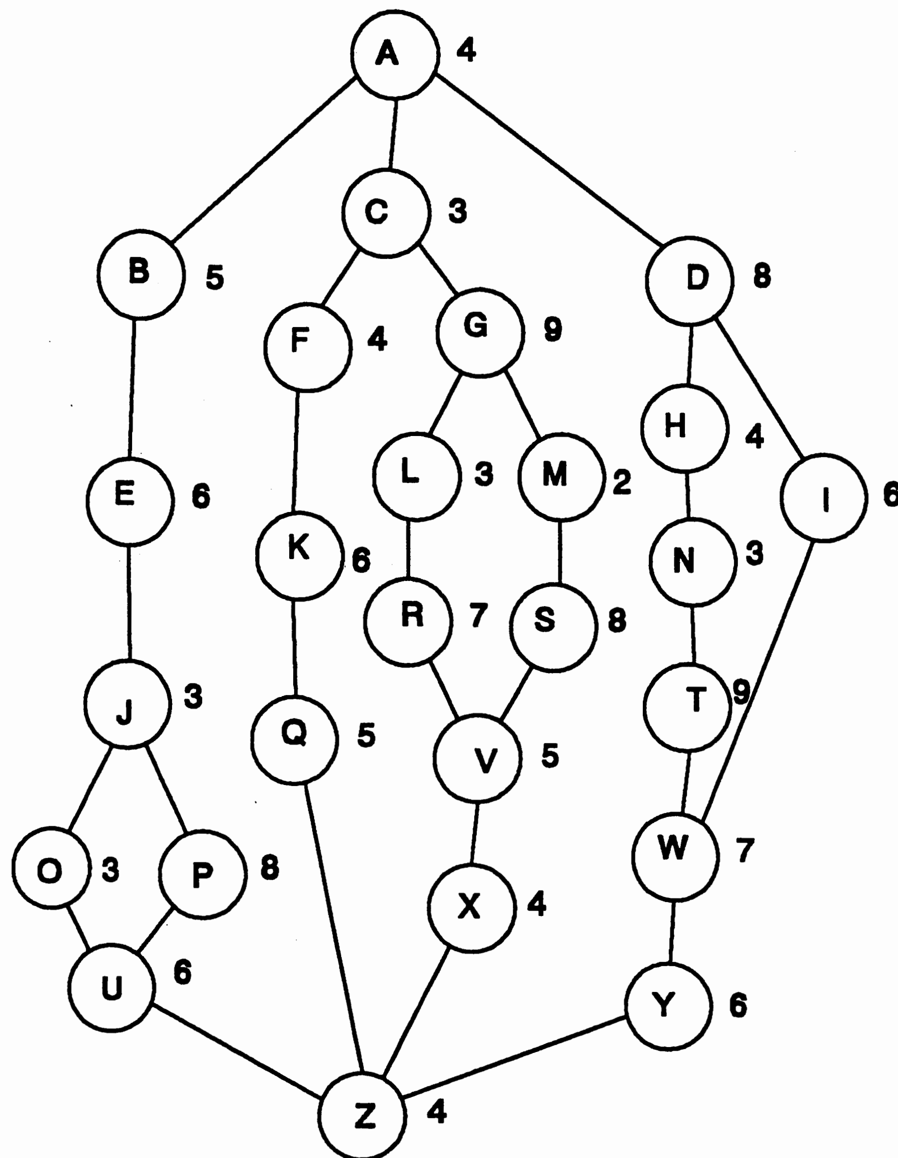


Figure 13. A task system represented as a weighted directed acyclic graph

5.5 ESP/VL: A Scheduling Algorithm for Scheduling Independent Task Layers

The scheduling approach, ESP/VL, described in this section uses the ESP algorithm presented in the previous section, and the Variant-Load algorithm described in Section 4.3. Variant-Load algorithm is a near-optimal algorithm that can be used for scheduling of independent serial tasks. The scheme presented in this section uses the partitions created by the ESP algorithm and schedules the tasks in the resulting layers using the Variant-Load algorithm. The tasks in each layer are scheduled in increasing order of enumeration of the indices assigned to the layers. That is, the tasks in layer l_i are scheduled (and must be finished) before the tasks in layer l_{i+1} start execution. A common problem associated with static scheduling of tasks in a multiprocessor environment arises with situations in which the actual processing time required by tasks is different from predicted ones. Typically, if the actual processing times are less than the predicted ones, the only undesirable outcome is wasted processor time and increased job flow time. However, if the actual time required for execution of a task is larger than the predicted value, then the problem of determinacy of the execution sequence of tasks must be handled in order to guarantee correct termination values. In other words, a mechanism for synchronization and delaying the execution of remaining tasks must be in place in order to guarantee program correctness. The scheduling scheme presented in this section is particularly useful for synchronization of the execution time of the tasks in each layer. In the event that one or more tasks in a given partition require longer processing times than anticipated, then the next scheduled partition will not be released for execution until the tasks in the currently running layer finish execution. This principle also holds in situations when the tasks in a layer finish execution before the expected time. In this case, the next batch of tasks constituting the next layer can start execution before the original release time. In order to provide such a synchronization mechanism, some form of run-time support is necessary. Such support could be

implemented in the form of special control code injected into the tasks at the time of compilation, or in the form of operating system run-time support. The first approach would be a less expensive one because of the lower overhead involved in comparison to the operating system support.

To demonstrate how the ESP/VL scheme for scheduling of task systems by using the Variant-Load and ESP algorithms works, the task system in Figure 13 will be used as an example in the following analysis. As shown in Section 5.4, applying ESP algorithm to the task system graph in Figure 13 yields eight independent layers, each with different number of elements. The width of a graph is defined to be equal to the number of elements in the largest layer. The example in Section 5.4 demonstrates that the width of the graph in Figure 13 is six, indicating that a maximum of six processors are necessary to run each task in parallel. In order to schedule this task system on a target machine, the number of processors available to a program needs to be considered. If there are as many processors as tasks in the largest layer, the task system can be scheduled on the target machine as partitioned by this algorithm. The length of the schedule in such a case is:

$$T_p = \sum_{i=1}^{k} \max_{1 \le l \le m_j} \{w(t_{i,l})\}$$

where k denotes the number of layers in a graph and m_j denotes the number of tasks in each layer j.

However, as the results of our performance evaluations will demonstrate in the next section, T_p will yield an undesirable value, in relation to the performance ratio, in the case of task graphs in which the number of tasks in each layer is less than or equal to twice the number of available processors.

Using the above formula, T_p = 53 for parallel execution of the task system in Figure 13. The sequential execution length for the task system is T_s = 138. The speed-up gained in this case, S, is approximately 2.6. Note that the critical path length in the graph

of Figure 13 is T_C = 45. Since no scheduling algorithm can yield a schedule length shorter than the critical path length, therefore T_{OPT} = 45 for this task system graph. Given T_{OPT}, the highest possible speed-up achievable in the execution, $S_{\max}$, assuming an unlimited number of processors, is approximately 3 for the task system graph in Figure 13.

The solution for scheduling a task system consisting of independent tasks on a limited number of processors was discussed in Section 4.3. Variant-Load algorithm can be used for scheduling of the tasks in each of the layers of a program graph. If the number of tasks in any layer is greater than the number of processors available, then Variant-Load algorithm can be applied to each such layer to produce a schedule for the given layer. Figure 14 shows the resulting schedule for graph of Figure 13 assuming that there are only three processors available.

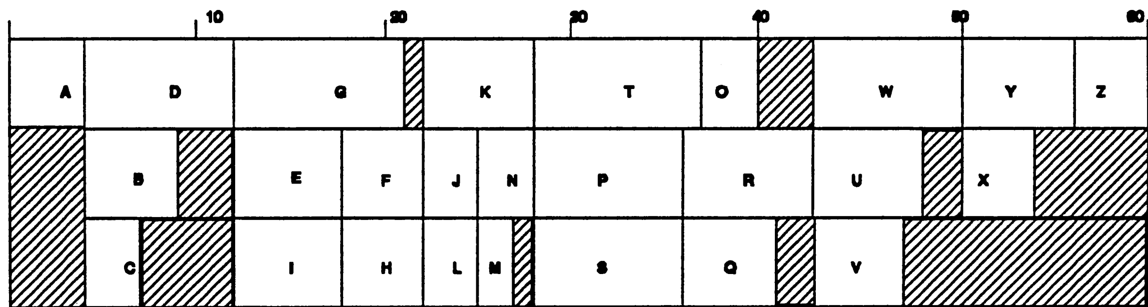


Figure 14. A Schedule for the task system in Figure 13 on three processors

As can be seen in Figure 14, the schedule length for p = 3 is T_p = 60. Variant-Load algorithm has been used for scheduling of the independent task sets {E, F, G, H, I}, {J, K, L, M, N}, and {O, P, Q, R, S, T} on three processors. As a result of decreasing the number of processors from six to three, the schedule length has increased by seven time

units. The scheduling of the task set {E, F, G, H, I} accounts for one unit of time increasing the length of the schedule for this set from nine to ten. The remaining six time units in increased schedule length are caused by scheduling of task set {O, P, Q, R, S, T}, whose schedule length increases form nine to fifteen time units, once scheduled on three processors. Scheduling of task set {J, k, L, M, N} on six processors incurs a schedule length of six which remains invariant when scheduled on three processors.

In the introduction section of this chapter, it was argued that, since our main goal in scheduling of jobs is achieving the highest possible speed-up, as long as there are sufficient number of processors available, no attempts are made to decrease the number of processors required, unless utilizing fewer number of processors does not increase the schedule length. For example, in Figure 14, scheduling the task set {J, K, L, M, N} on three and six processors resulted in no change in the schedule length.

In order to decrease the number of processors needed, an algorithm can be developed such that, before any actual processor assignments take place, Variant-Load algorithm is run on independent task sets with fewer than requested processors to determine whether decreasing the number of processors still yields the same schedule length. For example, scheduling the task set {P, P, Q, R, S, T} in Figure 13 with processing times {3, 8, 5, 7, 8, 9} on five processors gives the same schedule length as scheduling these tasks on six processors. Utilizing fewer processors than five increases the schedule length. Therefore, it can be concluded that the minimum number of processors for maximum speed-up in execution of the tasks in the task system graph of Figure 13 using the ESP/VL approach is five.

5.5.1 Performance Evaluation

To investigate the suitability of the approach described in the previous section, a number of experiments were performed in which the dependent variable was the number

of tasks in each layer. The measure of performance was defined to be the performance ratio defined in Section 1.6.

5.5.1.1 Design Methodology

Six experiments were performed in which there were four variables involved. The independent variables were the number of levels (layers) in a task system graph, the number of available processors, and the processing time of tasks. The dependent variable used was the number of tasks in each layer. The range of values used for the independent variables in all six experiments were constant. These values were defined as follows. The number of layers in each task graph ranged between 8 and 16, the range for the number of processors used was 16 to 32, and the task processing times were in the range 1 to 100. The number of tasks in each layer of a task system are defined as a multiple of the number of processors. For example, in the first experiment, the number of task in each layer was defined to be $n_i \leq p$, and in the second experiment this number was $2p \leq n_i \leq 4p$, where n_i represents the number of tasks at level i, $1 \leq i \leq k$, of a task system with k levels. Twenty task systems were generated and scheduled for each of the six experiments.

In order to establish the non-interference of the independent variables used, we also performed some experiments in which one or two of the independent variables were altered. For example, task graphs with fewer layers were generated and scheduled on a number of processors determined by a different range from the ones used in the reported experiments. The results were consistent with the ones in the six experiments reported in this section. Therefore, we conclude that using different sets of independent variables is not necessary because the ranges defined are scalable to problems of different size that are proportional to the ones used.

The purpose of these experiments was to establish a relationship between the task

graph size and the performance of the scheduling approach used. The performance measure used was $R_p(A)$ as defined in Section 1.6, which is the ratio of the performance of a given algorithm A, to the performance of an approximation for the same problem. The performance of our algorithm was taken to be the schedule length produced by our approach. The approximation of the schedule length used was determined by using the measure LB as defined in Section 5.2. The results of the experiment are discussed in the next section.

5.5.1.2 Simulation Results

The ESP/VL approach described in Section 5.5, divides a task graph into sets of independent layers and schedules the tasks in each layer as independent tasks, using the Variant-Load algorithm discussed in Section 4.3. In Section 5.5, we defined the schedule length T_p, to be equal to the sum of the largest task in each layer, in situations when there are enough processors for the tasks in each layer (i.e., number of available processors is equal to the width of the graph). However, this schedule length is not very desirable in situations where $T_c = T^*_{OPT} << T_p$. To establish a relationship between the number of available processors and the number of tasks in each layer of a task system, the experiments described in the previous subsection were conducted. The results of our experiments are reported in TABLE III. The performance curve is shown in Figure 15. As described in the methodology, the measure of performance used was the performance ratio $R_p(A)$. Ideally, we would want the performance ratio to be as close to 1 as possible. TABLE III shows the range of the values used for the dependent and independent variables defined in these experiments. Also shown in this table, is the average performance ratio for each of the six experiments. As can be seen in Figure 15, the performance of the ESP/VL approach is very poor for task systems in which the number of task in each layer is less than or equal to p. As the number of tasks in each layer is increased, the performance ratio exhibits a rapid improvement. For example, in

the first experiment, when $n_i \leq p$, the performance ratio exhibits a 73% inefficiency while in the third experiment, when $2p \leq n_i \leq 4p$, the same percentage goes down to 5% which is a rapid improvement. The results of these experiments demonstrate that the scheduling approach described in this section is particularly suitable for applications with a large number of tasks that must be run on a limited number of processors.

5.6 Ranked Weight Algorithm: A Heuristic Approach for Scheduling Dependent Task Systems

In this section, we present another scheduling approach for scheduling of dependent task systems. Similar to the approach presented in Section 5.5, the new algorithm discussed in this section first partitions the task system graph into a number of independent layers. However, Unlike the previous approach in which the tasks in layer l_i must finish execution before tasks in layer l_{i+1} could be initiated, this new approach selects tasks (for scheduling) based on weight assigned to each task. The *Ranked Weight* algorithm described in this section can be characterized as a list scheduling algorithm in which the urgency rule for scheduling or dispatching of tasks is determined based on certain criteria.

As described in the survey of related work in chapter 1, a majority of the scheduling algorithms found in the literature can be characterized as list scheduling algorithms. One of the best-known heuristic list scheduling algorithms for scheduling task systems that exhibit inter-task dependencies is the CP/MISF (Critical Path/Most Immediate Successors First) heuristic [Kasahara and Narita 84] which is motivated by Hu's labeling of tasks according to their distance from the root. Hu's method is known as the *Critical Path* approach. The CP/MISF heuristic combines the Critical Path approach and an additional heuristic which determines the priority of the tasks in a layer based on the

TABLE III
EXPERIMENTS WITH VARYING THE NUMBER OF TASKS

EXPERIMENTS	1	2	3	4	5	6
Levels	(8-16)	(8-16)	(8-16)	(8-16)	(8-16)	(8-16)
No. of Tasks per Level	p	2p	(2..4)p	(4..8)p	(8..16)p	(16..32)p
No. of Processors	(16..32)	(16..32)	(16..32)	(16..32)	(16..32)	(16..32)
Task Weights	(1..100)	(1..100)	(1..100)	(1..100)	(1..100)	(1..100)
Performance Ratio	1.73	1.29	1.05	1.017	1.004	1.001

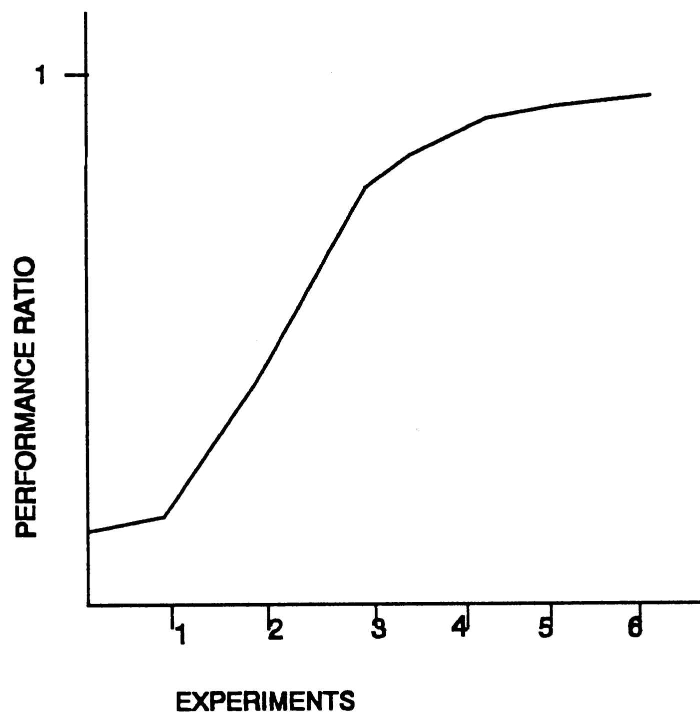


Figure 15. Performance of the ESP/VL scheduling approach

number of their immediate successors. Tasks with a large number of immediate successors have higher priority. This algorithm is claimed to have incorporated the best heuristic for scheduling of DAGs. Polychronopoulos [Polychronopoulos 88] argues that tasks can be characterized as either serial or parallel tasks (where serial tasks contain a single execution thread while parallel tasks may be composed of a number of independent serial tasks). Therefore, using the number of immediate successors might not be a realistic priority scheme. He proposes a parameterized heuristic for assignment of priorities to tasks in his algorithm using the following rules. He gives priorities to

- tasks that lie on the critical path,
- tasks with long execution times,
- tasks with largest number of immediate successors, and
- tasks with successors that have long processing times.

He then uses these rules to define three parameters that can be used for proportional allocation of processors to tasks. His allocation scheme assigns a single processor to each serial task. Any number of remaining processors are distributed to parallel tasks based on their priorities. The flexibility of Polychronopoulos' algorithm is due to the capability of altering the weight of the three parameters. By changing the relative weight of the parameters, the algorithm can assign different priority weights to the above stated rules.

5.6.1 The Ranked Weight Heuristic

It is demonstrated shortly that the rules described above [Polychronopoulos 88] are embedded in the Ranked Weight heuristic, presented in this section. The MISF heuristic in Kasahara and Narita's algorithm gives a higher priority to tasks with largest number of immediate successors. Unless it is used in the context of Hu's algorithm, where tasks have equal processing times, the disadvantage of the CP/MISF method is that the processing times of the tasks are not used as a factor for load balancing. The heuristic

used in the algorithm presented in this section takes into account the processing time required by a task and all its successors. For example, the processing time of the source node (assuming that the task system graph starts with a single node with no predecessors) will be equal to the total execution time of the entire task system. We refer to this number as the *ranked weight* of the task. Ready tasks are scheduled in the descending order of their ranked weights. This seems to be a stronger heuristic than the MISF heuristic because it selects the tasks that consume a larger amount of processor time through their successor path(s). This heuristic performs at least as well as the MISF heuristic because if a task with large number of successors also has a large ranked weight, it will be given a higher priority. On the other hand, however large the number of immediate successors of a task is, if the ranked weight of this task is less than that of some other ready task, this task will be given a lower priority (TABLE IV shows the expected processing time, $w(t_i)$, and the ranked weight, $rw(t_i)$, of the tasks in the task graph of Figure 8). Additionally, the Ranked Weight heuristic gives priority to the tasks that lie on the critical path of a graph in a special sense.

Giving priority to the tasks that lie on the critical path of a graph is an old heuristic that has been used by many [Baer 73] [Kwan et al. 91] [Polychronopoulos 88]. The Ranked Weight heuristic presented in this dissertation, looks at the critical path of a task graph in a new light. The rationale for giving priority to the critical path tasks in scheduling is meant to ensure that the execution of the tasks on the most expensive path of the graph will not "lag behind" such that their delayed execution will increase the schedule length.

The general approach for scheduling of task graphs is as follows. Given task graph G, ready tasks in G (tasks with no predecessors) are scheduled and conceptually erased from the graph. Lets call the resulting graph G' which differs from G by removal of the scheduled tasks. Referring to the task graph in Figure 8 as G, the first ready task that can

be selected for scheduling is the task with label A. The resulting task graph , G', after scheduling of A is shown in Figure 16. The critical path in graph G is the path that contains tasks with labels A, C, F, and I. Therefore, if an algorithm gives priority to the tasks that lie on the critical path, the above tasks will get the highest priority. However, after the task with task label C is scheduled, the new graph G'' (shown in Figure 17) has a different critical path, namely, the tasks that lie on the path B, E, H, I. The Ranked Weight heuristic takes advantage of this dynamically created critical path and gives priority to the tasks that define the new critical path. Using the Ranked Weight heuristic, the next task selected for scheduling (after tasks with labels A and C are scheduled) is task B and therefore, tasks whose early execution does not contribute to the significant progress are delayed until they become the highest priority task for execution at a later point.

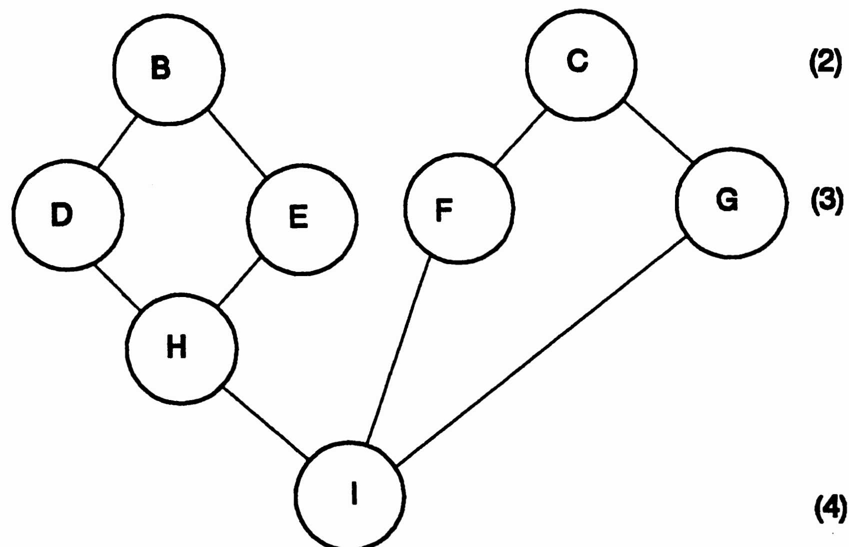


Figure 16. Task graph G' resulting from removal of task A in Figure 8

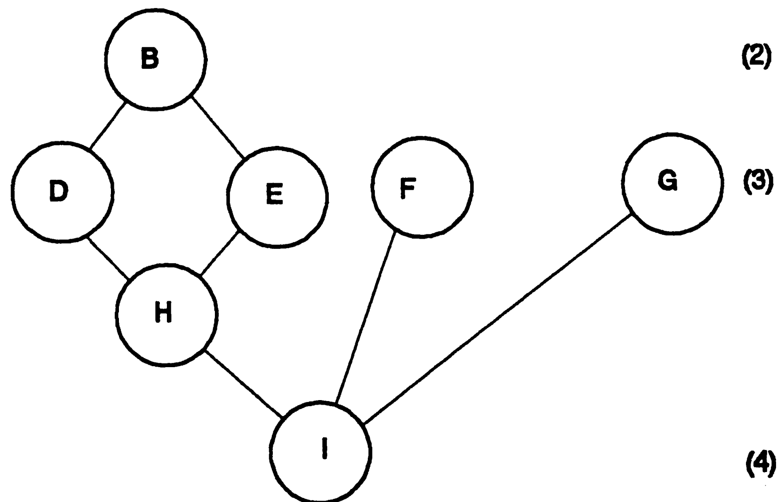


Figure 17. Task graph G'' resulting from removal of task C in Figure 16

5.6.2 The Ranked Weight Algorithm

In this section, an informal description of the Ranked Weight algorithm is presented and followed by several examples and a formal description. As discussed earlier, the labeling introduced in Hu's algorithm in which tasks are divided into levels according to their distance from the root, is known as the *Critical Path* or *Highest Level First* method. In the case of general precedence graphs, we refer to this approach as the Latest Schedule Partitioning (LSP). LSP produces layers of tasks according to the latest time a task can be scheduled (see for example, Figure 11 in Section 5.3). It is also possible to partition the task precedence graph according to a task's distance from the source. This will produce layers of tasks according to the earliest time a task can be scheduled. We refer to this partitioning as the Earlist Schedule Partitioning (ESP) (see Figure 12, Section 5.3). The partitioning used by the algorithm presented in this section is based on the ESP scheme.

The Ranked Weight algorithm operates by selecting a task that is at the highest level, has the largest ranked weight, and whose predecessors all have finished execution. The implicit priority rules embedded in the Ranked Weight algorithm are as follows. By selecting tasks that are at the highest level, this algorithm takes advantage of the ESP partitioning. By using the ranked weight of tasks, this approach has implicitly incorporated several of the priorities discussed previously (namely, giving priority to a) tasks with long execution times, b) tasks with largest number of immediate successors, and c) tasks with successors that have long processing times).

The Ranked Weight algorithm takes as input two adjacency lists (defined below), the ranked weight of each task (informally defined in the previous subsection and formally defined below), and the level associated with each task. This information can be extracted by a single pass through the graph. The ESP algorithm (presented in Section 5.4) was used and augmented with necessary parameters to supply the required input. The ESP algorithm uses the boolean connectivity matrix of the task graph (which is an upper triangular matrix representing a DAG). The successor and predecessor sets required by the ranked weight algorithm are readily available by the boolean matrix since the rows in this matrix correspond to the immediate successors and the columns correspond to the immediate predecessors of a given task. The ranked weights are calculated during the same pass that determines the task layers.

One of the required input adjacency lists identifies the immediate successor(s) and the other list identifies the immediate predecessor(s) of a task. The ranked weight, $rw(t_i)$, of task t_i is defined to be the sum of the weights of all descendants tasks of t_i plus the weight of task t_i defined as follows

$$rw(t_i) = w(t_i) + (\Sigma w(t_j), \quad \text{for all } t_j \in \{ \text{ descendants of } (t_i) \}).$$

The ranked weights are used as an urgency criterion for scheduling of the tasks.

Input: $List_{succ}$: The successor adjacency list.

$List_{pred}$: The predecessor adjacency list.

$Q[k]$: A multilevel list with k levels.

$level\ (t_i)$: The task system graph level associated with task t_i.

Output: An assignment of tasks to processors.

Method:

$Q[l] \leftarrow \phi, \ \forall \ 1 \le l \le k$ where k = number of levels in the graph.

$Q[1] \leftarrow Q[1] \cup \{source\ node\}$.

1) Let $l = \min\{$ level number, l', such that $Q[l'] \neq \phi$ $\}$,

$t \leftarrow$ head of $Q[l]$,

Schedule task t on the first, lowest indexed, available processor,

$Q[l] \leftarrow Q[l] - \{t\}$.

2) For all $t' \in List_{succ}[t]$ do

$List_{pred}[t'] \leftarrow List_{pred}[t'] - \{t\}$,

If $List_{pred}[t'] = \phi$ then

Insert t' in the sorted list $Q\ [\ level\ (t')\ ]$

according to $rw\ (t')$.

3) If $Q[l] = \phi, \ \forall \ 1 \le l \le k$, then HALT,

else GO TO Step **1**.

Figure 18. The Ranked Weight Algorithm

5.6.3 Algorithm Description

The predecessor adjacency list in this algorithm is used to identify the tasks that are ready to be scheduled. A task is *ready* when its predecessor set is empty. The successor adjacency list is used to identify the successors(s) of a task after its completion. The use of the successor adjacency list eliminates the search involved in identifying ready tasks. The urgency rule used in the scheduling of tasks involves the selection of the ready task that is at the highest level (highest non-empty subqueue) and has the highest ranked weight. Ties are broken by selecting the task that has the highest processing time. Ready tasks are added to the subqueue with a subqueue number that corresponds to the task level.

The Ranked Weight algorithm operates by threading the ready tasks (i.e., tasks whose predecessor sets are empty) to the appropriate subqueue. The tasks in subqueues are sorted based on the ranked weight of the tasks in each subqueue. Use of different subqueues cuts down on the cost of ordering the ranked weights. Initially, the multilevel queue Q, is empty. The Ranked Weight algorithm starts by placing the source node (the task with no predecessors) in the first subqueue, $Q[1]$. Step one of the algorithm finds the lowest indexed non-empty subqueue and schedules the task at the head of this subqueue (task t) on the lowest indexed available processors and deletes task t from the queue. Step two erases task t from the predecessor set of task t's successors. After this deletion, if any task t', ends up with an empty predecessor set (which means task t has been the only predecessor for t' at that point), then t' is added to the appropriate level in the Q that corresponds to the graph level for t', task t' is now ready. Steps 1 and 2 of the algorithm are repeated until all tasks are scheduled and $Q[l]$ is empty for all $1 \leq l \leq k$.

The following example helps clarify the algorithm presented in this section. Let us consider the precedence graph in Figure 8. Numbers in parentheses identify the level of

the tasks. TABLE IV shows the ranked weights $rw(t_i)$ and the expected processing times, $w(t_i)$ of each task. Note that the tasks appear in sorted order based on the ranked weight of the tasks.

TABLE IV
RANKED WEIGHT AND EXPECTED PROCESSING TIMES OF THE TASKS IN FIGURE 8

	A	C	B	E	D	H	F	G	I
$rw(t_i)$	76	42	34	27	26	24	18	12	4
$w(t_i)$	4	16	5	3	2	20	14	8	4

The successor and predecessor adjacency lists of the graph in Figure 8 are shown in Figure 19.

Initially, task A is added to subqueue one (the highest subqueue). After scheduling and completion of task A, the successors of A are identified through the successor list as tasks B and C, which serve as indices to the predecessor list. Task A is then deleted from the predecessor lists of tasks B and C. This leaves tasks B and C with empty predecessor lists and causes these tasks to be placed in subqueue two, $Q[2]$, as ready tasks. Notice that in the case of a task with multiple predecessors, the completion and subsequent removal of, say, either task D or E alone from the predecessor set of task H does not cause the placement of task H into the ready queue. The schedule corresponding to the

precedence graph of Figure 8 on two processors is shown in Figure 20. The schedule in Figure 20 yields a makespan of $T_p = 46$. It should be mentioned that the above algorithm does not guarantee optimality in all cases. However, it can yield optimal non-preemptive schedules in some cases and some vary close to optimal schedules in other cases. The shortest non-preemptive schedule length for the given graph is 45, with sequencing tasks (A, B, D, C, F, I) on one processor and (E, H, G) on the other.

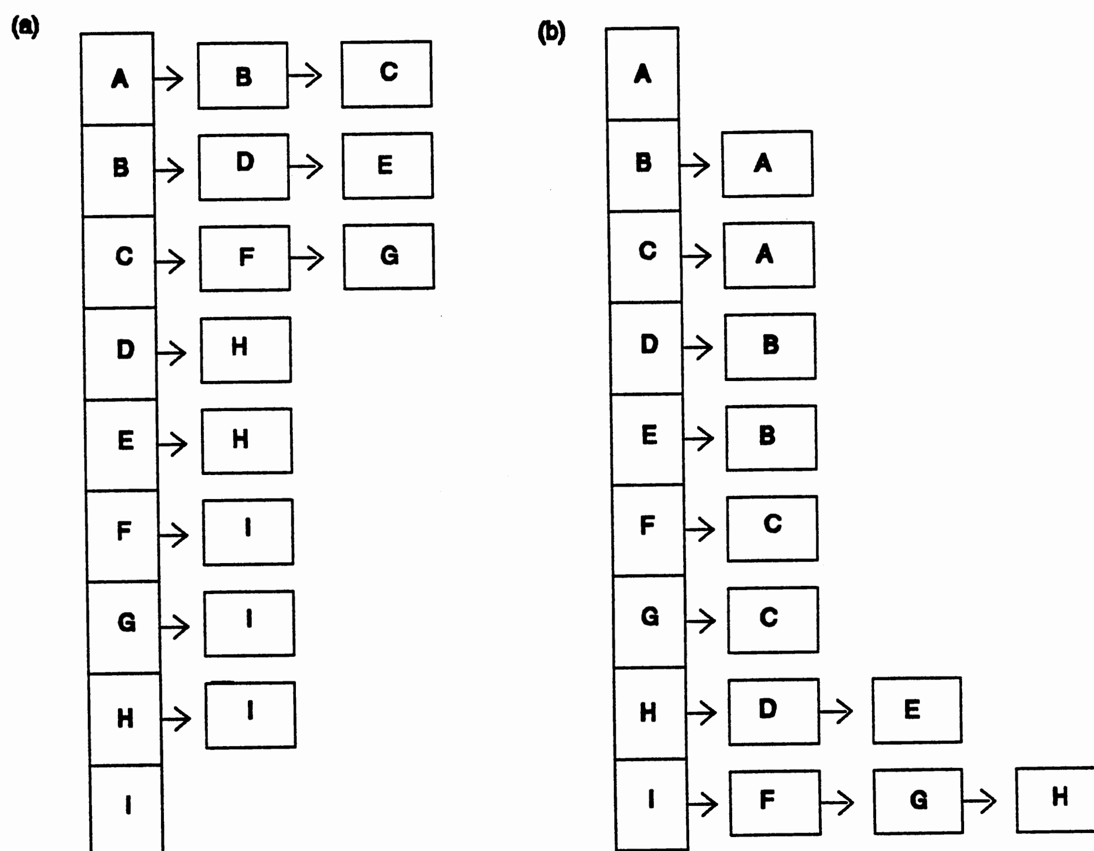


Figure 19. Successor list (a) and predecessor list (b)

Anomalies related to list scheduling algorithms were discussed in Section 1.4. We present a new anomaly in this type of scheduling in which the graph topology and the total processing time required by the graph is unchanged but the schedule length changes by rearranging some of the task processing times. Consider the task graph in Figure 8. By exchanging the processing times of tasks D and F and those of tasks E and G, the schedule length increases from 46 to 52. The schedule of the modified version of the task graph in Figure 8 in shown in Figure 21. This schedule is optimal for this particular task graph under non-preemption policy.

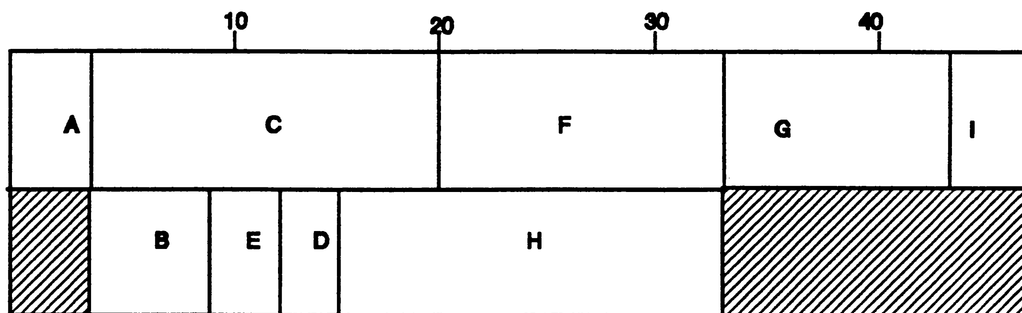


Figure 20. A schedule for the task system of Figure 8

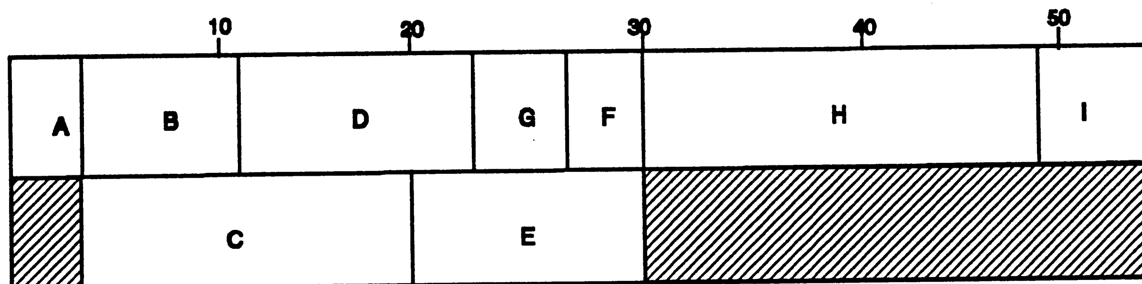


Figure 21. Schedule for the modified version of the task graph in Figure 8

Since finding the optimal schedule length on a case by case basis for a large number of task graphs is not possible, in order to evaluate the performance of the ranked weight algorithm, an approximation of the optimal schedule length is used in our evaluations. The approximation used was the lower bound on the schedule length, LB. The lower bound is calculated using the bounds defined in Section 5.2. For example, considering the task graph given in Figure 8 (whose task processing times are given in TABLE IV), the sequential execution time for this graph is T_s = 76 and its critical path is T_c = 38. Using the above data, LB = max{34, 30} + 8 = 42 for the task graph of Figure 8, scheduled on two processors. Notice that in general $T_{OPT} \geq LB$. The schedule length produced for the task graph of Figure 8 using the Ranked Weight algorithm is T_p = 46, using two processors. Manual examination of all possible schedules for this task graph yields an optimal schedule length of 45. However, since determining the optimal schedule length for arbitrarily large task graphs is not possible, we use an approximation of the optimal schedule length T^*_{OPT} and define it as $T^*_{OPT} = LB$. Using T^*_{OPT} as the optimal schedule length, the performance ratio $R_p(A)$, for the graph in Figure 8, using two processors is R_2(Ranked Weight) = 1.09 which indicates a nine percent error in the schedule length compared to the approximated optimal solution while the actual deviation from the optimal solution for this particular schedule is two percent when we consider T_{OPT} = 45 instead of the approximation of T^*_{OPT} = 42.

5.6.4 Performance Evaluation

In order to evaluate the performance of the Ranked Weight algorithm, performance modeling studies were conducted. A wide range of task system graphs were considered in this study that included randomized topologies as well as extreme cases involving different graph widths and heights. We define the *width* of a task graph to be the size of the independent task set (layer) with the largest number of tasks. The task graphs were generated by a parameterized algorithm that we developed for controlling the width and

height of the graph and the number of tasks in each layer. The maximum out-degree of a task could also be controlled in order to generate different graphs with different number of immediate successors. After these acyclic graphs are generated, they are converted to precedence graphs by removing the redundant paths. This is done to improve the run-time of processing the task graphs for the purpose of our simulation studies only, otherwise the produced schedules are not affected by removal of the redundant paths. Removal of the redundant paths did not relax any precedence constraints.

5.6.4.1 Design Methodology

Task graphs are generated using the approach described in the previous subsection. The generated graphs are partitioned into layers of independent serial tasks using the ESP algorithm (described in Section 5.4) augmented with parameters that calculate the ranked weight of tasks while partitioning the task system.

Two experiments evaluate the performance of the Ranked Weight algorithm. In one experiment, the number of available processors is set to the width of the task system graph. In the second experiment, the number of available processors is defined by a random number as $2 \leq p \leq width$. There are six different groups of tasks involved in each of the experiments. The first three groups involved task graphs with a number of tasks ranging in 5 to 20 for the first group, 21 to 40 for the second group, and 41 to 60 for the third group. The last three groups consisted of task graphs with a fixed number of tasks each. The performance of the produced schedules was measured using the performance ratio $R_p(A)$, defined in Section 1.6.

5.6.4.2 Simulation Results

In each of the two experiments, 300 task graphs are generated and scheduled for each of the six different groups of task graphs. The first column in TABLE V and VI shows the range of the number of tasks in each task group. The numbers in parentheses

represent the average number of tasks over all 300 task graphs generated in that group. Columns 2 through 6 show the performance of the algorithm in comparison to T^*_{OPT} where ε represents the error rate. None of the 3600 task graphs scheduled had an error rate of more than 25 percent compared to T^*_{OPT}.

The rows in TABLE V and VI represent the degree of optimality of the produced schedules. For example, in TABLE V, the row corresponding to task graphs with 5 to 20 tasks, 287 or 95.67 percent of the 300 task graphs produce an optimal schedule and 8 or 2.67 percent of the produced schedules are less than or equal to five percent longer than the optimal schedule. In the case of task graphs with 100 tasks each, 39.67 percent of the produced schedules indicate an optimal schedule length and 38 percent of the schedules are within five percent of the error bound. The last row in each table shows the average performance of all 1800 task graphs scheduled according to the error bounds in each column.

TABLE V
RESULTS OF 300 RUNS WHEN THE NUMBER OF PROCESSORS $P = WIDTH$

Number of tasks n (Avg)	$\varepsilon=0$ Number of cases (%)	$0.0\leq\varepsilon\leq0.05$ Number of cases (%)	$0.05\leq\varepsilon\leq0.10$ Number of cases (%)	$0.10\leq\varepsilon\leq0.20$ Number of cases (%)	$0.20\leq\varepsilon\leq0.25$ Number of cases (%)
5-20(12)	287(95.67)	8(2.67)	3(1.00)	1(0.33)	1(0.33)
21-40(31)	229(76.33)	42(14.00)	18(6.00)	11(3.67)	0
41-60(57)	174(58.00)	85(28.33)	26(8.67)	13(4.33)	2(0.67)
80	129(43.00)	114(38.00)	35(11.67)	20(6.67)	2(0.67)
100	119(39.67)	141(47.00)	28(9.33)	12(4.00)	0
200	63(21.00)	166(55.33)	25(8.33)	45(15.00)	1(0.33)
Total %	55.61%	30.89%	7.5%	5.66%	0.33%

TABLE V shows the results of the first experiment in which $p = width$ of the graph. The number of cases listed in the column where $\varepsilon = 0$ indicate the cases that yielded an optimal schedule. As can be seen from this table, 55.61 percent of the 1800 task graphs scheduled produced an optimal schedule. If an error bound of less than or equal to ten percent is considered as an acceptable solution, then the degree of success of the produced schedules rises to 94 percent (although in statistics, a 5% error bound is defined to be the acceptable deviation, we believe it is permissible to relax this bound to 10% because of the pessimistic approximation used in our measurements). In the case of the groups of task graphs with a fixed number of tasks, the success rate was consistent with the overall performance. For example, for the task graphs consisting of 100 tasks each, 96 percent of the produced schedules are within 10 percent error from the optimal solution.

The lower bound, LB, used for measuring the degree of optimality of the produced schedules in this research is achievable under preemptive scheduling. The Ranked Weight algorithm is a non-preemptive scheduling algorithm and therefore produces optimal schedules that in some cases are larger than LB. In other words, the produced schedules for many of the cases that exhibit a larger error may indeed be the best non-preemptive schedule that can be produced.

The results reported in TABLE VI show the case in which $2 \leq p \leq width$ of the graph. The results of the second experiment are less attractive than the first experiment in which $p = width$. However, these results are consistent with the performance of the best heuristic algorithm for scheduling of dependent task systems, CP/MISF [Kasahara and Narita 84] in which similar experiments using fewer than the required number of processors produced longer schedule lengths compared to schedules produced using an unlimited number of processors.

TABLE VI
RESULTS OF 300 RUNS WHEN THE NUMBER OF PROCESSORS $2 \leq P \leq WIDTH$

Number of tasks n (Avg)	$\varepsilon=0$ Number of cases (%)	$0.0 \leq \varepsilon \leq 0.05$ Number of cases (%)	$0.05 \leq \varepsilon \leq 0.10$ Number of cases (%)	$0.10 \leq \varepsilon \leq 0.20$ Number of cases (%)	$0.20 \leq \varepsilon \leq 0.25$ Number of cases (%)
5-20(12)	192(64.0)	38(12.67)	28(11.00)	33(11.00)	9(3.00)
21-40(31)	91(30.33)	67(23.00)	69(23.00)	69(23.00)	4(1.33)
41-60(57)	50(16.67)	93(31.00)	102(34.00)	53(17.67)	2(0.67)
80	47(15.67)	109(36.33)	96(32.00)	48(16.00)	0
100	41(13.67)	119(39.67)	93(31.00)	46(15.33)	1(0.33)
200	14(4.67)	111(37.00)	63(21.00)	111(37.00)	1(0.33)
Total %	24.16%	29.83%	25.05%	20.00%	0.94%

TABLE VI shows that a total of 53.99 percent of the cases are either optimal solutions or are within five percent from the defined lower bound. If the acceptable error bound is relaxed to ten percent, then the success rate goes up to 79.04 percent of cases. The Ranked Weight algorithm produces very good schedules when there are enough processors to run the tasks. In extreme cases that involve task systems with very large number of tasks, it was shown that the ESP/VL approach presented in Section 5.5 works best.

5.7 Independent Path Scheduler

Two different approaches are presented in this dissertation so far that can be used for scheduling of dependent task systems. Both of these approaches (Ranked Weight and

ESP/VL) first divide the task system graph into layers of independent serial tasks. Strengths of each approach and their suitability for particular scheduling problems were discussed earlier. Both ESP/VL and the Ranked Weight algorithms are suitable for scheduling of task systems in a shared memory environment because no special efforts are made in determining a particular processor to which a task is mapped. Therefore, the communication cost arising from the random assignment of tasks to processors (in the presence of intertask dependencies) in a distributed or private memory machine can offset the gain in employing multiple processors for executing a job.

This section presents a new approach for partitioning of task system graphs called *Vertical Partitioning* that can be used both for distributed and shared memory environments. The objective of the Vertical Partitioning approach is to a) determine the minimum number of processors necessary for maximum speed up and b) an assignment and mapping of tasks to processors such that the communication cost is minimized. The program representation used in Vertical Partitioning is the control flow graph model. Tasks in this model are defined as sequential blocks of code with no branching into or out of them except at the beginning and/or at the end. Each task also has a clearly defined input and output. Given such a model, the number of independent paths or *threads* of execution represent the maximal degree of parallelism. This number is referred to as the *nullity* of the graph and is denoted by N. The nullity of a graph is defined as $e-n+2$, where e represents the number of the edges and n represents the number of nodes in a graph [Berge 73] [Temperly 73]. Therefore, N represents the number of processors necessary. However, we show that after devising a schedule for a given task system, it is possible to optimize the number of required processors such that $N \leq p$.

Consider the task graph in Figure 13. The number of edges in this graph is $e = 31$ and the number of nodes is $n = 26$. Using the formula $N = e - n + 2$, there are seven independent paths in the graph of Figure 13. It is clear that these are not a set of unique

paths. Depending on the Vertical Partitioning approach used, different sets of independent paths can be identified. We refer to each independent path as a vertical partition (as opposed to "horizontal partitions" or "task layers" discussed in Section 5.4). Each vertical partition may be viewed as a *thread* of execution. In scheduling the resulting threads on a machine, the ideal situation is for the target machine to have as many processors as the nullity of the graph corresponding to the program under consideration. One possible vertical partitioning of the task graph in Figure 13 and its schedule on seven processors is shown in Figure 22. It is worth mentioning that with availability of the required number of processors, the schedule length will be the same regardless of how the task graph is "sliced" vertically. Under such a condition, the schedule length for vertical partitioning is $T_p = T_c = T_{OPT}$ and the shortest possible schedule length is achieved. However, this assertion is true if the relative weight of the communication costs at certain fork and join points within the graph are ignored. We concentrate on this aspect shortly.

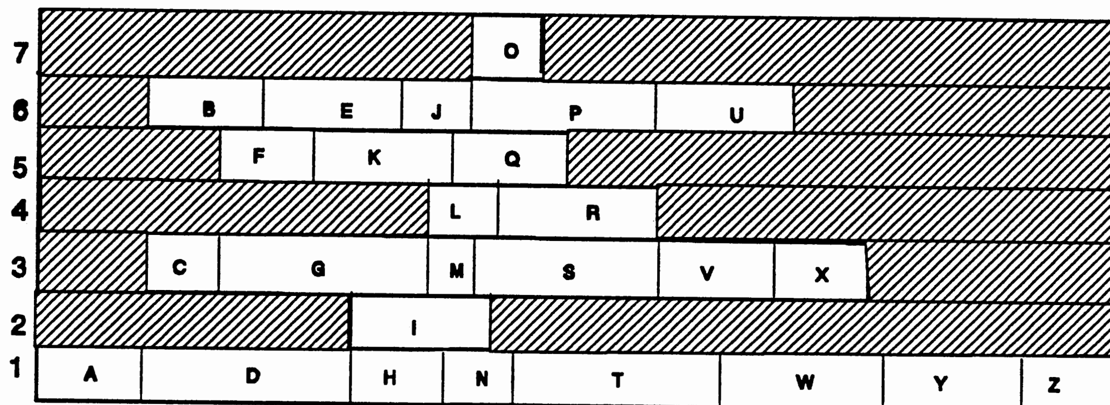


Figure 22. Vertical Partitioning of the task system graph in Figure 13

One simple optimization that can be done before a mapping of tasks to processors takes place is to reduce the number of required processors if possible. For example, for the schedule shown in Figure 22, it is possible to map the workload assigned to processors 2 and 7 onto one processor. Therefore, the minimum number of processors required for maximum speed up in scheduling of this task system graph is six. The schedule length under this condition is the optimal schedule $T_{OPT} = T_C = 45$. Further optimization of the number of required processors is possible by considering the precedence constraints in scheduling of the tasks in each thread. The new schedule for task graph in Figure 13 (on six processors) is shown in Figure 23.

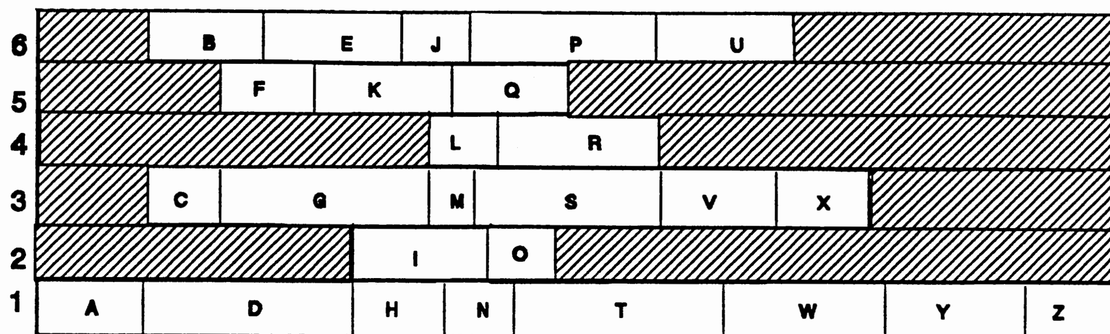


Figure 23. A schedule for the task system graph in Figure 13 on six processors

The communication overhead in vertical partitioning is only critical at two specific points in the task system graph. We refer to these points as Fork and Join latches (F&J latches). F-latches involve those tasks in the graph that have a fan-out degree of greater than one. Analogously, J-latches involve those tasks that have a fan-in degree of greater than one. For example, in the task graph of Figure 13, the task with label A is an F-latch of degree three and the task with label Z is a J-latch of degree four. Associated with each edge in the graph is a cost that corresponds to the communication cost between the pair of tasks that it connects.

In order to determine the independent threads of execution in a task graph, these communication costs will have to be considered at F&J latches. For example in Figure 13, lets suppose that task G is the task at the head of a new thread. In order to determine whether to choose tasks L and R or tasks M and S for the thread headed by F-latch G, the weight of the out-going edges at the F-latch will have to be considered. Assuming that the edge (G, L) is the more expensive edge, then Task M will itself be designated as the first task of a new thread. This type of partitioning minimizes the communication overhead.

It is possible that the number of available processors for scheduling of a task system is less than the nullity of the task graph. There are two possibilities under such a condition. One possibility is to attempt to schedule the tasks such that the execution threads are scheduled on the available processors without increasing the schedule length (for example, see the schedules in Figures 22 and 23). If scheduling of the task system without increasing the schedule length beyond the critical path length is not possible, then multiple threads must be mapped to the same processor and scheduled sequentially. Obviously, under such a condition we would like to provide a mapping that minimizes the communication overhead. In order to assign more than one thread to the same processor, we will have to concentrate on the F&J latches that minimize the communication cost and at the same time increase the schedule length the least. As describe in the beginning of this section, vertical partitioning can also be used for shared memory machines effectively where many of the issues discussed about minimizing the communication overheads are not a major concern.

CHAPTER VI

SUMMARY AND CONCLUSIONS

6.1 Introduction

To speed up the execution of computer programs, different portions of the program must be executed on separate processors. The problem of identifying parallel processable components of a program is referred to as *program partitioning*. Each identified partition, called a *task*, must be executed on a separate processor. Multiprocessor scheduling is concerned with sequencing and scheduling of tasks on processors with the objective of creating schedules with the shortest possible length. Task systems can be divided into two classes of dependent and independent tasks. Scheduling of independent tasks is concerned with producing shortest possible schedule length without any consideration given to the order of execution of the tasks constituting a task system. Dependent task system scheduling also is concerned with producing schedules with shortest possible length however, in this type of scheduling, the precedence constraints between pairs of tasks must be considered.

The main objective of this dissertation research is to investigate and consider a variety of issues related to multiprocessor scheduling and to propose solutions for the scheduling problem.

6.2 Summary

The general problem of multiprocessor scheduling is a combinatorial optimization problem and thus belongs to the family of NP-complete problems. Similar to other

problems belonging to the NP family, in order to solve the scheduling problem we must resort to approximation algorithms that use heuristics for producing near-optimal solutions. A survey of exact and heuristic multiprocessor scheduling algorithms is presented through out this dissertation. Most exact algorithms that produce optimal schedules owe their optimality to the restrictive constraints imposed on the task system and/or the machine characteristics. Major characteristics of some of the major scheduling algorithms reviewed as well as the algorithms developed in this dissertation are shown in TABLE VII.

TABLE VII
CHARACTERISTICS OF SEVERAL MULTIPROCESSOR SCHEDULING ALGORITHMS

	No. of Processors		Task Weight		Precedence			Preemption		Optimal	
Algorithm	2	>=2	1	>=1	0	Tree	General	Yes	No	Yes	No
1		X		X	X				X		X
2		X		X	X				X		X
3		X		X	X				X		X
4		X		X	X				X		X
5		X	X			X			X	X	
6	X			X			X	X		X	
7	X		X				X		X	X	
8		X		X		X	X		X		X
9		X		X		X	X		X		X
10		X		X			X		X		X

Algorithms: 1) Variant-Load
2) LPT [Johnson et al. 74]
3) D&F [Polychronopoulos 86]
4) Multifit [Coffman et al. 78]
5) [Hu 61]
6) [Muntz and Coffman 69]
7) [Coffman and Graham 72]
8) Ranked Weight
9) ESP/VL
10) CP/MISF [Kasahara and Narita 84]

Computer programs were represented using graph models. A variety of graph topologies were considered and solutions which yielded acceptable schedules were presented and discussed. A near optimal scheduling algorithm is developed that can be used for scheduling of independent serial tasks. Scheduling of dependent tasks systems is less trivial than scheduling of independent tasks. Dependent task systems are studied and analyzed extensively. Three different approaches are presented for scheduling of dependent task systems. Factors that affect the quality of such schedules include the general task graph topology such as task graph height and width, number of tasks in each graph level, and the task processing times. Machine characteristics that affect the schedule length include the number of available processors and the particular nature of the implementation platform. Two scheduling algorithms are developed that can be used for task system scheduling for shared memory machines. A third approach is presented for scheduling of dependent task systems in a private memory environment that minimizes the communication costs and the number of required processors while maximizing the speed up of execution.

The performance of the developed algorithms is compared to the best known algorithms in the literature through simulation studies. It is shown that the developed algorithms in this dissertation research do at least as well as the best-known algorithms with a significantly lower run-time complexity. Another significant algorithm developed in this research is an algorithm that can be used for identifying independent task sets in a task system graph. The complexity of this algorithm is considerably better than the best known algorithm for solving the same problem.

6.3 Contributions

The current research has made a number of significant contributions to the area of multiprocessor scheduling. These contributions include the development of numerous algorithms for scheduling of dependent and independent task systems that do as well as

or better than the best known algorithms while improving the complexity of the existing algorithms. The ESP algorithm used for partitioning of the task system graphs has a worst case complexity of $O(n^2)$ compared to the best known algorithm, MBFS, that has complexity $O(n^3)$. The Ranked Weight algorithm developed in this research, for scheduling of dependent task systems, demonstrate a performance comparable to the best known algorithms in terms of the optimality of the produced schedules. The complexity of the Ranked Weight algorithm is $O(npk)$ where k is the number of layers in the graph while its best known counter part, CP/MISF, has a run time complexity of $O(n^2 + pn)$.

An algorithm is developed for scheduling of independent serial tasks, Variant-Load algorithm. The performance of this algorithm is compared to three other algorithms, LPT, Multifit, and D & F. The performance of this algorithm is at least as good as the three algorithms it is compared to. The complexity of the Variant-Load algorithm is $O(np^2)$ in the worst case while its counter part, D & F has a complexity of $O(\log_2 (n/p))$ for the first phase and $O(n^2/p^2)$ for its optimizing phase.

A new task graph partitioning approach is developed which takes the communication cost of task executions into consideration and is therefore suitable for distributed environments. This approach can determine the minimum number of processors necessary for maximum speed up.

6.4 Future Work

A number of new and promising ideas have been developed and presented in this dissertation. The basic foundation for most of the future work has been established. The scheduling algorithms presented in this dissertation use the task processing times for creating static schedules. A major disadvantage in devising static schedules arises in situations in which the actual processing time of tasks at run time differ from the predicted ones. Under such conditions mechanisms for processor synchronization are

necessary in order to ensure program correctness. One component of the future work for this research involves developing strategies that allow for processor synchronization at run time for the developed algorithms. It has been already shown how such synchronization can be performed for the ESP/VL approach in which tasks are scheduled in separate waves and therefore run-time synchronization becomes quite manageable. We are interested in developing strategies for maintaining the precedence relationships between the tasks when using the Ranked Weight approach, in order to provide processor synchronization.

As mentioned above, the Ranked Weight algorithm is developed for creating static schedules. However, this algorithm could be adapted such that the compile time estimates are used for determining the ranked weights while the actual dispatching of tasks take place at run time. Under this new technique the tasks whose predecessors finish execution will enter a scheduling queue. The ranked weights can now be used as a priority for run-time scheduling. There are two feasible approaches to implement this scheme. One approach involves providing operating system support to perform the task of dispatching of ready tasks. The second approach involves embedding synchronization constructs in tasks at the time of compilation. This approach is referred to as *auto-scheduling* or self-scheduling [Polychronopoulos 88]. Refinements needed for developing such a priority based scheduler include investigation of the overhead involved in operating system intervention if operating system support is employed for the implementation of this scheme, and developing synchronization mechanisms and constructs for auto-scheduling for compiler support.

The current research has concentrated on optimizing schedules at the program level. No attempts have been made at the global level in order to increase the processor utilization. When executing a devised schedule for a task system, it is very likely that some of the processors allocated to a job at time t_0 will be idle until a later time t_i. The

same scenario holds for processors that are released at time t_j while the last processor to finish execution will not finish until time t_k, $j < k$.

It would be helpful to consider a meta-scheduler that attempts to optimize the system-wide schedule by trying to fit the "jagged ends" of the individual job schedules together and/or switch the assigned work to different processors in order to maximize processor utilization and increase the system throughput.

REFERENCES AND SELECTED BIBLIOGRAPHY

[Agrawal and Jagadish 88] P. Agrawal and H. V. Jagadish, "Partitioning Techniques for Large-Grained Parallelism," *Proceedings of the 7th International Phoenix Conference on Computers and Communication*, Mar. 1988, Scottsdale, AZ., pp. 31-38.

[Allen and Kennedy 82] J. R. Allen and K. Kennedy, "PFC: A Program to Convert Fortran to Parallel Fortran," Technical Report MASC-TR82-6, Rice University, Houston, TX, March 1982.

[Amdahl 67] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Computer Science Conference*, Vol. 30, 1967.

[Ammarguellat 90] Z. Ammarguellat, "A Control-Flow Normalization Algorithm and its Complexity," Technical Report CSRD-1024, *Center for Supercomputing Research and Development*, University of Illinois, Urbana, IL. Jun. 1990.

[Axford 89] T. Axford, *Concurrent Programming: Fundamental Techniques for Real-Time and Parallel Software Design*, John Wiley & Sons Ltd., Chichester, England, 1989.

[Bacon and Storm 91] D. F. Bacon and R. E. Storm, "Optimistic Parallelization of Communicating Sequential Processes," *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Williamsburg, VA, 1991, pp. 155-166.

[Baer 73] J. L. Baer, "A Survey of some Theoretical Aspects of Multiprocessing," *Computing Survey*, Vol. 5, No. 1, March 1973, pp. 31-80.

[Berge 73] *Graphs and Hypergraphs*, Amsterdam, The Netherlands, North-Holland, 9173.

[Bernstein 66] A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Transactions on Computers*, Vol. EC-15, No. 5, Oct. 1966, pp. 757-762.

[Bodin 83] L. Bodin, "Routing and Scheduling of Vehicles and Crews," *Computers and Operations Research*, Vol. 10, No. 2, 1983.

[Brown 71] A. R. Brown, *Optimum Packing and Depletion*, Elsevier Press, New York, 1971.

[Burke and Cytron 86] M. Burke, and R. Cytron, "Inter-procedural Dependence Analysis and Parallelization," *Proceedings of the 1986 Compiler Construction Conference*, August 1986, pp. 52-64.

[Carey 89] G. F. Carey (Ed.), *Parallel Supercomputing: Methods, Algorithms and Applications*, John Wiley & Sons Ltd., Chichester, England, 1989.

[Cheng 89] T. C. E. Cheng, "A Heuristic for Common Due Date Assignment and Job Scheduling of Parallel Machines," *Journal of the Operational Research Society*, No. 40, 1989, pp. 1129-1135.

[Cheng and Sin 90] T. C. E. Cheng and C. C. S. Sin, "A State-of-the-Art Review of Parallel-Machine Scheduling Research," *European Journal of Operational Research*, Vol 47, Elsevier Science Publishers, North-Holland, 1990, pp. 217-292.

[Coffman 66] E. G. Coffman, "Stochastic Models of Multiple and Time Shared Computer Operations," Ph.D. Dissertation, Department of Electrical Engineering, University of California, Los Angeles, CA. 1966.

[Coffman 67] E. G. Coffman, "Bounds on Parallel Processing of Queues with Multiple Jobs," *Naval Research Logic Quarterly*, Vol. 14, Sept. 1967, pp. 345-366.

[Coffman 76] E. G. Coffman (Ed.), *Computer and Job-Shop Scheduling Theory,* John Wiley and Sons, New York, 1976.

[Coffman and Denning 73] E. G. Coffman and P. J. Denning, *Operating Systems Theory*, Prentice-Hall Publishing Co., N. J., 1973.

[Coffman and Graham 72] E. G. Coffman, and R. L. Graham, "Optimal Scheduling for Two-Processor Systems," *Acta Informatica*, Vol. 1, No. 3, 1972, pp. 200-213.

[Coffman et al. 84] E. G. Coffman, M. R. Garey, and D. S. Johnson, "Approximation Algorithms for Bin-Packing - An Updated Survey," in *Algorithm Design for Computer System Design*, G. Ausiello, M. Lucertini, and P. Serafini (Eds.), Springer-Verlag New York, 1984, pp. 49-106.

[Coffman and Gilbert 85] E. G. Coffman and E. N. Gilbert "On the Expected Relative Performance of List Scheduling." *Operations Research*, No. 33, 1985, pp. 548-561.

[Coffman et al. 78] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "An Application of Bin-Packing to Multiprocessor Scheduling," *SIAM Journal of Computing*, Vol. 7, No. 1, Feb. 1978, pp. 1-17.

[Conway 63] M. Conway, "A Multiprocessor System Design," *Proceedings of the AFIPS Fall Joint Computer Conference*, 1963, pp. 139-146.

[Conway et al. 67] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*, Addison-Wesley Publishing Co., Reading, MA, 1967.

[Cook 71] S. A. Cook, "The Complexity of Theorem-Proving Procedures," *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, 1971, pp. 151-158.

[Cytron et al. 90] R. Cytron, J. Ferrante, and V. Sarkar, "Experiences Using Control

Dependence in PTRAN," in *Languages and Compliers for Parallel Computing*, D. Gelernter, A. Nicolau and D. Padua (Ed.), The MIT Press, Cambridge, Mass. 1990, pp. 186-412.

[Davis 73] E. W. Davis, "Project Scheduling under Resource Constraints - Historical Review and Categorization of Procedures," *AIIE Transactions*, Vol. 5, No. 4, 1973, pp. 297-313.

[De and Morton 80] P. De and T. E. Morton, "Scheduling to Minimum Makespan on Unequal Parallel Processors," *Decision Science*, No. 11, 1980, pp. 586-602.

[de Bakker 89] J. W. de Bakker (Ed.), *Languages for Parallel Architectures: Design, Semantics, Implementation, Models*, John Wiley & Sons Ltd., Chichester, England, 1989.

[Dennis 68] J. B. Dennis, "Programming Generality, Parallelism and Computer Architecture," *Proceedings of the IFIP Congress 1968*, North Holland, Amsterdam, The Netherlands, pp. 1-7.

[Desrochers 87] G. R. Desrochers, *Principles of Parallel and Multiprogramming*, Intertext Publishing Co., McGraw-Hill Publishing Co., New York, N.Y., 1987.

[Dogramasi and Surkis 79] A. Dogramasi and J. Surkis, "Evaluation of a Heuristic for Scheduling Independent Jobs on Parallel Identical Processors," *Management Science*, No. 23, 1979, pp. 1208-1216.

[Dreifus 58] P. Dreifus, "System Design of the GAMMA-60 Computer," *Proceedings of the 1958 Western Joint Computer Conference*, Spartan Books, New York, pp. 130-133.

[Dror and Stern 87] M. Dror and H. I. Stern, "Parallel Machine Scheduling: Processing Rates Dependent on Number of Jobs in Operation," *Management Science*, No. 33, 1987, pp. 1001-1009.

[Dumond and Mabert 88] J. Dumond and V. A. Mabert, "Evaluating Project Scheduling and Due date Assignment Procedures: An Experimental Analysis," *Management Science*, Vol 34, No. 1, Jan. 1988, pp. 101-118.

[Ein-Dor 85] P. Ein-Dor, "Grosch's Law Revisited," *Communications of the ACM*, Vol. 28, No. 2, Feb. 1985, pp. 142-151.

[Eisemann 57] K. Eisemann, "The Trim Problem," *Management Science*, No. 3, 1957, pp. 279-281.

[Emrath 89] P. Emrath, "Program Laborious," *UNIX Review*, April 1989, pp. 51-60.

[Emrath et al. 88] R. Emrath, D. Padua, and P. C. Yew, "Cedar Architecture and its Software," Technical Report No. 796, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1988.

[Estrin and Turner 63] G. Estrin and R. Turner, "Automatic Assignment of Computations in a Variable Structure Computer System," *IEEE Transactions on Electronic Computers*, Vol. EC-12, Dec. 1963, pp.756-773.

[Feitelson and Rudolph 90] D. G. Feitelson and L. Rudolph, "Distributed Hierarchical Control for Parallel Processing," *Computer*, Vol. 23, No. 5, May 1990, pp. 65-78.

[Ferrante et al. 83] J. K. Ferrante, K. Henstein, and J. Warren, "The Program Dependence Graph and its Uses in Optimization," *IBM Technical Report RC 10208*, August 1983.

[Flynn 72] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, Vol. C-21, No. 9, Sept. 1972, pp. 948-960.

[Franklin 78] M. A. Franklin, G. S. Graham, and R. K. Gupta, "Anomalies with Variable Partition Paging Algorithms," *CACM*, Vol. 21, No. 3, 1978, pp. 232-236.

[Gehani and McGettrick 88] N. Gehani and A. D. McGettrick, *Concurrent Programming*, Addison Wesley Publishing Co., Reading, MA, 1988.

[Gill 58] S. Gill, "Parallel Programming," *Computer Journal*, Vol. 1, April 1958, pp. 2-8.

[Gilmore and Gomory 61] P. C. Gilmore and R. E. Gomory "A Linear Programming Approach to the Cutting Stock Programs," *Operations Research*, Vol. 9, 1961, pp. 849-859.

[Girkar and Polychronopoulos 88] M. Girkar, and C. Polychronopoulos, "Partitioning Programs for Parallel Execution," *Proceedings of 1988 ACM International Conference on Supercomputing*, St. Malo, France, July 1988, pp. 216-229.

[Glenbe 89] E. Glenbe, *Multiprocessor Performance* John Wiley & Sons Ltd., Chichester, England, 1989.

[Gonzalez 77] M. J. Gonzalez, "Deterministic Processor Scheduling," *ACM Computing Survey*, No. 9, 1977, pp. 173-204.

[Gonzalez and Ramamoorthy 71] M. J. Gonzalez and C. V. Ramamoorthy, "Program Suitability for Parallel Processing," *IEEE Transactions on Computers*, Vol. C-20, June 1971, pp. 647-654.

[Gosden 66] J. A. Gosden, "Explicit Parallel Processing Description and Control in Programs for Multi- and Uni-Processor Computers," *Fall Joint Computer Science Conference*, Vol. 29, 1966, pp. 651-660.

[Graham 69] R. L. Graham, "Bounds on Multiprocessor Timing Anomalies," *SIAM Journal of Applied Mathematics*, Vol. 17, No. 2, Mar. 1969, pp. 416-429.

[Graham 76] R. L. Graham, *Bounds on the Performance of Scheduling Algorithms, Computer and Job/Shop Scheduling Theory*, (E. G. Coffman, Ed.), John Wiley, New York, 1976.

[Guan and Langston 91] X. Guan and M. A. Langston, "Time-Space Optimal Merging and Sorting," *IEEE Transactions on Computers*, Vol. 40, No. 5, May 1991, pp. 596-602.

[Ha and Lee 91] S. Ha and E. A. Lee, "Compile-Time Scheduling and Assignment of Data-Flow Program Graphs with Data-Dependent Iterations," *IEEE Transactions on Computers*, Vol. 40, No. 11, Nov. 1991, pp. 1225-1238.

[Hirschberg 82] D. S. Hirschberg, "Parallel Graph Algorithms without Memory Conflicts," *Proceedings of the 20th Allerton Conference*, 1982, pp. 257-263.

[Hockney and Jesshope 81] R. W. Hockney and C. R. Jesshope, *Parallel Computers*, Hilger Press, Bristol, 1981.

[Hu 61] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, Vol. 9, Nov.-Dec. 1961, pp. 841-848.

[Hwang and Briggs 84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Publishing Co., New York, N.Y., 1984.

[Hwang and DeGroot 89] K. Hwang and D. DeGroot, *Parallel Processing and Supercomputers and Artificial Intelligence*, McGraw-Hill Series in Supercomputing and Parallel Processing, McGraw-Hill Publishing Co., New York, N.Y., 1989.

[Johnson 73] D. S. Johnson, "Near-Optimal Bin Packing Algorithms," *Doctoral Dissertation*, MIT, Cambridge, MA., 1973.

[Johnson et al. 74] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms," *SIAM Journal of Computing*, Vol. 3, No. 4, December 1974, pp. 299-326.

[Kalos 87] M. H. Kalos "Monte Carlo Methods and the Computers of the Future," *Supercomputers, Algorithms, Architectures, and Scientific Computation*, Edited by F. A. Matsen and T. Tajima, University of Texas Press, 1987.

[Karin and Smith 87] S. Karin and P. N. Smith, *The Supercomupter Era*, Hartcourt Brace Jovanovich Publishers, Boston, MA.,1987.

[Karp and Flatt 90] A. H. Karp and H. P. Flatt, "Measuring Parallel Processing Performance," *CACM*, Vol. 33, No. 5, May 1990, pp. 539-543.

[Kasahara and Narita 84] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Transactions on Computers*, Vol. C-33, No. 11, Nov. 1984, pp. 1023-1029.

[Keller 70] R. M. Keller, "On Maximally Parallel Program Schemata," *Proceedings of the 11th Symposium on Switching and Automata Theory*, 1970, pp. 33-50.

[Kuck 78] D. J. Kuck, *The Structure of Computers and Computations*, John Wiley & Sons, New York, NY, 1978.

[Kuck et al. 80] D. Kuck, R. Kuhn, B. leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Proceedings of the 4th International Computer Software and Application Conference*, October 1980, pp. 709-715.

[Kuck, et al. 86] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, "Parallel Supercomputing Today and the Cedar Approach," *Science*, Vol. 231, No. 4740, Feb. 1986, pp. 967-974.

[Kwan et al. 90] A. W. Kwan, L. Bic, and D. D. Gajski, "Improving Parallel Program Performance Using Critical Path Analysis," In *Languages and Compilers for Parallel Computing*, D. Gelernter, and A. Nicolau (Eds.), The MIT Press, 1990, pp. 358-373.

[Lea 87] R. M. Lea, "An Overview of the Influence of Technology on Parallelism," *Major Advances in Parallel Processing*, Edited by C. Jesshope, 1987, pp. 3-12.

[Li and Yew 88] Z. Li and P. C. Yew, "Interprocedural Analysis for Parallel Computing," Technical Report No. 734, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1988.

[Li and Yew 1990] Z. Li and P. C. Yew, "Some Results on Exact Data Dependence Analysis," in *Languages and Compliers for Parallel Computing*, D. Gelernter, A. Nicolau and D. Padua (Ed.), The MIT Press, Cambridge, Mass. 1990, pp. 374-401.

[Lo et al. 90] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. Mohamed, and J. Telle, "ORIGAMI: Software Tool for Mapping Parallel Computations to Parallel Architectures," Technical Report CIS-TR-89-18, Department of CIS, University of Oregon, Eugene, OR. Jan. 1990.

[Long and Clarke 89] D. L. Long and L. A. Clarke, "Task Interaction Graphs for Concurrency Analysis," *Proceedings of the 11th International Conference on Software Engineering*, 1989, pp. 44-52.

[McGreary and Gill 89] C. McGreary and H. Gill, "Automatic Determination of Grain Size for Efficient Parallel Processing," *CACM*, Vol. 32, No. 9, Sept. 1989, pp. 1073-1078

[McNaughton 59] R. McNaughton, "Scheduling with deadlines and Loss Functions'" *Management Science*, Vol. 6, Jan. 1959, pp.1-12.

[Midkiff et al. 1990] S. P. Midkiff, D. A. Padua, and R. Cytron, "Compiling Programs with User Parallelism," in *Languages and Compliers for Parallel Computing*, D. Gelernter, A. Nicolau and D. Padua (Ed.), The MIT Press, Cambridge, Mass. 1990, pp. 402-422.

[Minsky 70] M. Minsky, "Form and Computer Science," ACM Turing Lecture, *JACM*, Vol. 17, No. 2, February 1970, pp. 197-215.

[Muntz and Coffman 69] R. R. Muntz and E. G. Coffman, "Optimal Multiprocessor Scheduling on Two-Processor Systems," *IEEE Transactions on Computers*, Vol. C-18, No. 11, Nov. 1969, pp. 1014-1020.

[Muraoka 71] Y. Muraoka, "Parallelism Exposure and Exploitation in Programs," Ph.D. Dissertation, Department of Computer Science, University of Illinois, Urbana-Champaign, IL. 1971.

[Neilsen 85] T. N. Neilsen, "Combinatorial Bin Packing Problems," *Doctoral Dissertation*, The University of Arizona, Tucson, AZ, 1985.

[Noronha and Sarma 91] S. J. Noronha and V. V. S. Sarma, "Knowledge-Based Approach for Scheduling Problems: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, No. 2, Jun. 1991, pp. 160-171.

[Murgolo 85] F. D. Murgolo, "Approximation Algorithms for Combinatorial Optimization Problems," *Doctoral Dissertation*, University of California at Irvine, 1985.

[Peterson 85] V. L. Peterson, "Use of Supercomputers in Computational Aerodynamics," *Proceedings of the 1985 Science and Engineering Symposium*, Cray Research Inc., Minneapolis, 1985.

[Polychronopoulos 86] C. D. Polychronopoulos, "On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems," Ph.D. Dissertation, Computer Science Department, University of Illinois at Urbana-Champaign, Champaign, IL, 1986.

[Polychronopoulos 88] C. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishing, Norwel, MA, 1988.

[Qin et al. 91] B. Qin, H. A. Sholl, and R. A. Ammar, "Micro Time Cost Analysis of Parallel Computations," *IEEE Transactions on Computers*, Vol. 40, No. 5, May 1991, pp. 613-628.

[Ramamoorthy 66] C. V. Ramamoorthy, "Analysis of Graphs by connectivity considerations," *JACM*, Vol. 13, No. 2, April 1966, pp. 211-222.

[Ramamoorthy and Gonzales 69] C. V. Ramamoorthy and M. J. Gonzalez, "A Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," *Proceedings of the AFIPS Fall Joint Computer Conference*, 1969, pp. 1-15.

[Ramanan 84] P. V. Ramanan, "Topics in Combinatorial Algorithms," *Doctoral Dissertation*, University of Illinois, Urbana, IL, 1984.

[Rothkopf 66] M. A. Rothkopf, "Scheduling Independent Tasks on Parallel Processors," *Management Science*, Vol. 12, Jan 1966, pp. 437-447.

[Russel 69] E. C. Russel, Automatic Program Analysis," Ph. D. Dissertation, Department of Electrical Engineering, University of California, Los Angeles, CA., 1969.

[Sahni 76] S. Sahni, "Algorithms for Scheduling Independent Tasks," *JACM*, No. 23, 1976, pp. 116-127.

[Sahni 77] S. Sahni, "General Techniques for Combinatorial Approximation," *Operations Research*, Vol. 27, 1977, pp. 920-927.

[Saltz et al. 91] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-Time Parallelization and Scheduling of Loops," *IEEE Transactions on Computers*, Vol. 40, No. 5, May 1991, pp. 603-612.

[Samadzadeh 91] Farideh A. Samadzadeh, "Implementation of Cooperating and Competing Algorithms on Sequent S81," *The 29th ACM Southeast Regional Conference*, Auburn, Alabama, April 1991, pp. 356-358.

[Samadzadeh and Hedrick 91a] F. Samadzadeh and G. E. Hedrick, "Near-Optimal Multiprocessor Scheduling," *The Proceedings of The 1992 ACM Computer Science Conference* , Kansas City, MO. 1992, pp. 477-484 .

[Samadzadeh and Hedrick 91b] F. Samadzadeh and G. E. Hedrick, "A Heuristic Multiprocessor Scheduling Algorithm for Creating Near-Optimal Schedules Using Task System Graphs," *The Proceedings of The 1992 ACM Symposium on Applied Computing*, Kansas City, MO. 1992, pp. 711-718 .

[Sarkar 91] V. Sarkar, "PTRAN: The IBM Parallel Translation System," in *Parallel Functional Languages and Compilers*, B. K. Szymanski (Ed.), Addison-Wesley Publishing Co., Reading, MA, 1991.

[Seiworek 89] D. Sieworek, "Complex Tasks, Higher Powers," *UNIX Review*, April 1989, pp. 40-49.

[Sevcik 89] K. C. Sevcik, "Characterization of Parallelism in Applications and Their Use in Scheduling," *Performance Evaluation Review*, Vol. 17, No. 1, May 1989, pp. 171-180.

[Shang and Fortes 91] W. Shang and J. A. B. Fortes, "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies," *IEEE Transactions on Computers*, Vol. 40, No. 6, Jun. 91, pp. 723-742.

[Shen et al. 90] Z. Shen, Z. Li, and P. C. Yew, "An Empirical Study of Fortran Programs for Parallelizing Compilers," Technical Report No. 983, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, April 1990.

[Skedzielewski and Glauert 85] S. Skedzielewski and J. Glauert, "IF1 - An Intermediate Form for Applicative Languages," *Manual M-170*, Lawrence Livermore National Laboratory, Livermore, CA. July 1985.

[Skvarcius and Robinson 86] R. Skvarcius and W. B. Robinson, *Discrete Mathematics with Computer Science Applications*, Benjamin/Cummings Publishing Co. Melno Park, CA, 1986.

[Temperly 81] H. N. V. Temperly, *Graph Theory and Applications*, Ellis Horwood Series in Mathematics and Its Applications, England, 1981.

[Terrano, et al. 89] A. E. Terrano, S. M. Dunn, and J. E. Peters, "Using and Architectural Knowledge Base to Generate Code for Parallel Computers," *CACM*, Vol. 32, No. 9, Sept. 1989, pp. 1065-1072.

[Towsley et al. 90] D. Towsley, C. G. Rommel, and J. A. Stankovic, "Analysis of Fork-Join Program Response Time on Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 3, July 1990, pp. 286-303.

[Trelevin 90] P. C. Trelevin, Ed., *Parallel Computers: Object-Oriented, Functional, Logic*, John Wiley and Sons, New York, N. Y., 1990.

[Uht 91] A. K. Uht, "A Theory of Reduced and Minimal Procedural Dependencies," *IEEE Transactions on Computers*, Vol. 40, No. 6, Jun. 91, pp. 681-692.

[Uht et al. 87] A. K. Uht, C. D. Polychronopoulos, and J. F. Kolen, "On the Combination of Hardware and Software Concurrency Extraction Methods," Technical Report No. 694, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1987.

[Ullman 67] S. D. Ullman, "Complexity of Sequencing Problems," in *Computers and Job-Shop Scheduling*, E. G. Coffman, Ed., John Wiley and Sons, N. Y., 1967.

[Volansky 70] S. A. Volansky, "Graph Model Analysis and Implementation of Computational Sequences," Ph.D. Dissertation, Department of Electrical Engineering, University of California, Los Angeles, CA. 1970.

[Waltz 87] D. L. Waltz, "Applications of the Connection Machine," *Computer*, Vol, 20, No. 1, Jan. 1987, pp. 85-99.

[Wang and Gannon 89] K. Wang and D. Gannon, "Applying AI Techniques to Program Optimization for Parallel Computers," In *Parallel Processing for Supercomputers and Artificial Intelligence*, K. Hwang and D. DeGroot (Eds.), McGraw-Hill Series in Supercomputing and Parallel Processing, McGraw-Hill Publishing Co., New York, N.Y., 1989.

[Warshall 62] S. Warshall, "A Theorem on Boolean Matrices," *JACM*, Vol. 3, No. 1, Jan. 1962, pp. 11-12.

[Wilhelmson 87] R. Wilhelmson, (Ed.), *High Speed Computing: Scientific Applications and Algorithm Design*, University of Illinois Press, 1987.

[Williams 89] S. A. Williams, *Programming Models for Parallel Systems*, John Wiley & Sons Ltd., Chichester, England, 1989.

[Wolfe 82] M. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Dissertation, Department of Computer Science, University of Illinois, Urbana-Champaign, IL., Report No. UIUDCS-R-82-1105, 1982

[Wouk 86] A. Wouk (Ed.) *Parallel Processing and Medium-Scale Multiprocessing* Siam, Philadelphia, PA, 1986.

VITA

Farideh Ansari-Jafari Samadzadeh

Candidate for the Degree of

Doctor of Philosophy

Thesis: SCHEDULING ALGORITHMS FOR PARALLEL EXECUTION OF COMPUTER PROGRAMS

Major Field: Computer Science

Biographical:

Personal Data: Born in Tehran, Iran, January 22, 1957.

Education: Bachelor of Arts in Public Relations and Social Affairs from the College of Mass Communication, Tehran, Iran in May 1979; Master of Science in Interpersonal and Intercultural Communication from University of SW Louisiana, Lafayette, Louisiana in December 1982; Master of Science in Computer Science from The Center for Advanced Computer Studies, University of SW Louisiana, Lafayette, Louisiana in May 1987; completed the requirements for the degree of Doctor of Philosophy at Oklahoma State University in July 1992.

Professional Experience: Instructor of English as a second language, Shokouh's English Institute, Tehran, Iran, December 1975 to July 1979; Teaching Assistant, Communication Department, University of SW Louisiana, Lafayette, Louisiana, August 1980 to May 1982; Teaching/Research Assistant, The Center for Advanced Computer Studies, University of SW Louisiana, August 1986 to May 1987; Lecturer/Instructor of Computer Science, Computer Science Department, Oklahoma State University, Stillwater, Oklahoma, January 1988 to June 1992, Assistant Research Computer Scientist, Computer Science Department, Oklahoma State University, Stillwater, Oklahoma, July 1992 - present.

Professional Organizations: Voting member of the Association of Computing Machinery (ACM), member of the Scientific Research Society Sigma Xi.