4-1-1999

A parallel scheduling approach for multiprocessor scheduling.

Jianguo Cai

A Parallel Scheduling Approach for Multiprocessor Scheduling

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirement for the Degree

Master of Science

University of Nebraska at Omaha

by

Jianguo Cai

April 1999

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

THESIS ACCEPTANCE

Acceptance for the faculty of the Graduate College, University of Nebraska, in partial fulfillment of the requirements for the degree (Master of Science), University of Nebraska at Omaha.

Committee

Hesham Ali Computer Science

Hesham El-Rewini Computer Science

Azad Azadmanesh Azadmanesh 4-19-99, CS dept.

Peter Wolcott 4.19.99, Dept. of ISQA

Chairperson(s) Hesham Ali & Hesham El-Rewini

Date 4/19/99

A PARALLEL SCHEDULING APPROACH

FOR MULTIPROCESSOR SCHEDULING

Jianguo Cai, MS

University of Nebraska, 1999

Advisors: Hesham H. Ali and Hesham El-Rewini

A scheduling problem is an issue that needs to be addressed whenever a need is present to arrange a set of tasks into a set of processing units, under certain policies, with different possible outcomes. In general, the time complexity of finding an optimal solution for scheduling problem is exponential. However, in many situations, finding an optimal solution for a scheduling problem is essential. This necessity for efficient scheduling has spurred much research work in this area. As a result, many efficient heuristic scheduling algorithms have been developed. For the most part, however, these scheduling algorithms apply to parallel programs, the algorithms themselves are sequential and, as yet, little work has been done on paralleling scheduling algorithms.

In this thesis, we study the issue of parallel scheduling and present a new parallel scheduling scheme. The primary objective is to study parallel scheduling algorithms by comparing their performances with sequential scheduling algorithms. In this study, horizontal scheme, vertical scheme, as well as a new scheme, hybrid scheme, are implemented, and compared via simulation. The results of the conducted experiments show that horizontal scheme algorithms normally produce shorter schedules, while the

vertical scheme algorithms have better speedups. It also shows that the hybrid scheme achieves better parallelism, while still producing acceptable schedule length by producing schedules with the advantages of both horizontal and vertical schemes.

ACKNOWLEDGEMENTS

Words are not good enough to express my gratitude to Prof. Hesham H. Ali for his encouragement, understanding, patience, and sincere efforts. Without his help, this thesis could not be finished. Special thanks are also given to Prof. Hesham El-Rewini for his great help.

Also I want to thank the other committee members, Prof. Azad Azadmanesh, Prof. Peter Wolcott, for their encouragement and helps during my work.

At last, I want to thank my parents for their tremendous supports.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

The scheduling of dependent tasks in a distributed system is an important and challenging research area. In recent years, a lot of research effort has been directed towards finding better scheduling solutions. A majority of the existing algorithms are sequential themselves, even though they are used to solve parallel computing problems. A key issue is how fast can we complete the scheduling process in some specific scheduling applications. Parallel scheduling algorithms can speedup scheduling by making use of multi-processors in parallel machines.

1.1 Overview of Scheduling Problems

In our daily lives, we may encounter scheduling problems in many situations. The simplest example is that of an individual's daily time schedule or perhaps a project manager's project schedule. We also need time schedules for classroom use for courses in all schools. Also, in airports, flight scheduling is an essential issue to keep the whole system running efficiently and safely. In general, the scheduling problem is an issue that needs to be addressed when the need is present to arrange a set of tasks into a set of processing units under certain policies with different possible outcomes.

Today, the scheduling problem is gaining increasing attentions by many researchers from different fields like mathematics, computer science, business, and operation research. Many studies have been done for different versions of scheduling problems that deal with

dependent or independent tasks, connected or unconnected systems, single processor or multi-processors, etc. In this thesis, we study parallel scheduling algorithms for scheduling dependent tasks onto multi-processors machine.

Finding optimal schedules is an important and challenging problem in parallel and distributed computing. The development of effective techniques for the distribution of the processes of a parallel program on multiple processors, to optimizing given criteria, is a major area of research in parallel and distributed computing. The problem is to schedule the processing among process elements to achieve some performance goals, such as minimizing communication delays and execution time and maximizing resource utilization. The scheduling problem is challenging since it has been proven to be a NP-complete problem in its general form. We cannot get a guaranteed optimal schedule in polynomial time without any restriction, hence, only restricted algorithms can be found in the literature. Recently, some restricted forms of the scheduling problem have been studied and some heuristics scheduling algorithms have been introduced to solve many general versions of the problem in polynomial time. Heuristic methods rely on the rule of thumb to guide the scheduling processes in the proper track for a near optimal solution. For example, in list scheduling, each task is assigned a priority, then a list of tasks is constructed in decreasing priority order. When a processor is available, a ready task with the highest priority is selected from the list and assigned to that processor [7]. A heuristic algorithm can solve a scheduling problem in less than exponential time and still get a near optimal solution.

1.2 Motivations

The primary motivation for this thesis is the efficiency of scheduling. The speed of a schedule is a very important factor as well. For example, in a large international airport, the computer system should be able to reschedule all pending flights in a short period of time when unexpected events happen, for instance, a severe snowstorm. All operation systems for multi-processors systems need schedulers to allocate computing resources for running processes. The efficiency of these schedulers is very critical to the system because they affect the performance of whole systems and they need to be repeatedly run with high frequency. Clearly, parallelizing the schedulers provides a creative and efficient way of speeding up the scheduling process.

Another reason for parallel scheduling is to utilize multi-processor machines. Parallel and distributed computing is on the lower portion of a logistics growth curve, leading to mainstream acceptance of the ideas of parallelism by the late 1990s [7]. After decades of development, more powerful parallel supercomputers were made by different vendors. In 1993, Cray introduced Cray T-90, which was a vector supercomputer. It had 2 gigaflops per processor and had a 32 processor maximum, and a 2 nanosecond clock (500MHz). Recently, NEC produced the Cenju-3; it contains up to 256 VR4400SC processors connected by an Omega network. To make use of these powerful supercomputers, high performance software, including scheduling applications, for these parallel machines is needed. At the same time, the hardware prices are very low today. Multi-processor machines are now available and affordable for business, research, and other purposes.

Network of PC cases also form a distributed system that can be utilized in solving large tasks.

Although many scheduling algorithms are applied to parallel programs, the algorithms themselves are sequential and designed to be executed on a single processor system. Little work has been done in designing parallel algorithms for task scheduling. In this thesis, we study the designing issues of parallel scheduling algorithms to find a way to speedup scheduling in multi-processor machines.

1.3 Static Scheduling

According to how much task information is available before execution, a scheduling algorithm can be classified into one of the three categories: static scheduling, dynamic scheduling, or hybrid dynamic/static scheduling.

In static scheduling, all information about tasks and the relations among them is known before execution time and the assignment of the processors is done before program execution begins. The purpose here is to minimize the execution time of current programs while minimizing the communication delays. Since the optimal scheduling problem is NP-complete, research in static scheduling is focused on special cases and on heuristic solutions. For some special cases with some restrictions, optimal schedules can be found in polynomial time. Now, we know polynomial algorithms can be obtained in the following cases: (1) when the task graph is a tree [16]; (2) when the task graph is an interval order [11]; and (3) when there are only two processors available [22]. In the three

cases, all tasks are assumed to have the same execution time. In general, the heuristic approach can be classified into two categories. The first category is simple heuristic algorithms, which rely on the rules of thumb of heuristics to guide the schedule process. The most popular scheduling algorithm in this category is the list scheduling algorithm. The other category consists of algorithms that are more computationally intensive, but are not exponential, such as simulated annealing methods, mathematical programming methods, state space search methods, and generic methods.

On the contrary, in dynamic scheduling, the topologies among tasks and execution and communication costs are not available before the program executes. Conditional branches and loops are two program constructs that may cause non-determinism [7]. Finally, the hybrid dynamic/static scheduling is a combination of static and dynamic scheduling. In this thesis, we focus on the static scheduling problem exclusively.

There are also some other scheduling taxonomies, like preemptive scheduling versus non-preemptive scheduling, adaptive scheduling versus non-adaptive scheduling [7]. The non-preemptive scheduling means a task cannot be interrupted once it has begun execution. The non-adaptive scheduling means a scheduler does not change its behavior according to feedback from the system. For more information on scheduling taxonomies, please refer to [7].

1.4 Project Description

For some scheduling problems, the solutions for them can be used for a long period of time. For instance, the scheduling of classrooms for courses can be used for several months. In some cases solutions can be reused, like scheduling in a manufacturing plant. However, in some circumstance there are other kinds of scheduling problems that need to be solved in a very short period of time with high frequency of execution, for instance, the scheduler of an operating system for a parallel machine. So the efficiency of scheduling is very important here. One way to get higher efficiency of scheduling is to parallelize the scheduling algorithm itself by making use of the multi-processors of parallel computers. The purpose of parallel scheduling algorithms is to schedule faster, as well as to utilize the hardware resources of parallel computers.

We know there are two major domains in a scheduling algorithm. One is source domain and the other is target domain [2]. As illustrated in figure 1.1, the source domain can be viewed as the input of scheduling problems. It contains the information about the scheduling problems, such as tasks that need to be scheduled. Tasks can be represented by a macro data flow graph. The target domain is the schedule for target processing units. It's the output of scheduling algorithms. In sequential scheduling algorithms, only one node is scheduled into a target domain each time. The parallel scheduling algorithms schedule several nodes at the same time within different processing units. The quality and speed of a parallel scheduler depends on data partitioning [2].

We use three parallel scheduling schemes in this paper to study parallel scheduling, the vertical scheme, the horizontal scheme, and the hybrid scheme. Figure 1.1 illustrates the horizontal scheme and the vertical scheme [26]. The target processors are domains that tasks are scheduled onto. The physical processors refer to the actual processors in parallel machines. In horizontal schemes, the data flow graph is divided into several partitions

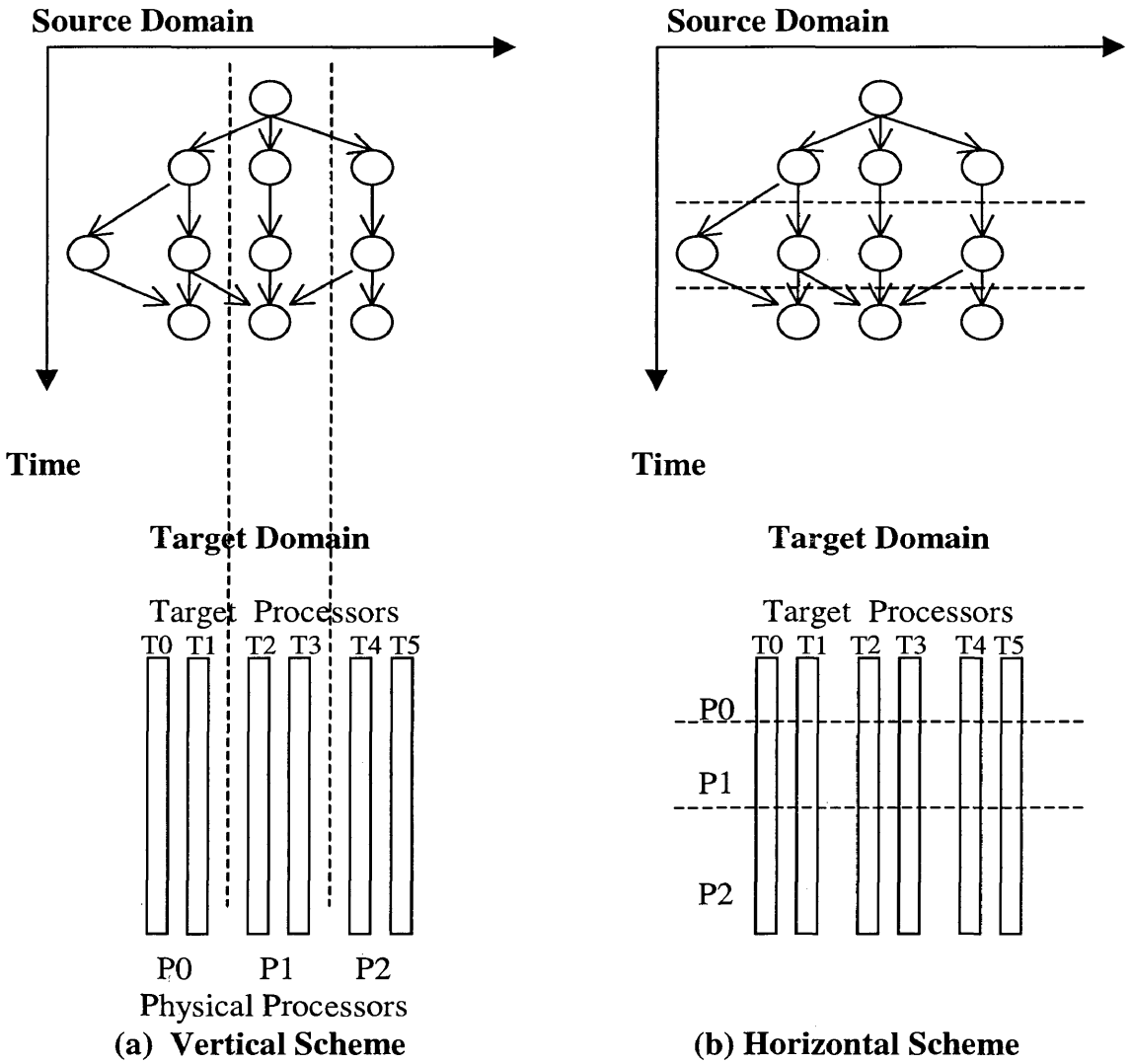


Figure 1.1 Vertical and Horizontal Schemes

using some kind of partition algorithm. Then, each partition is scheduled simultaneously using a sequential scheduling algorithm and a sub-schedule is acquired for each partition. The last step is to combine these sub-schedules into one final schedule. In a vertical

scheme, we just parallelize the sequential scheduling algorithm. Instead of scheduling a node each time, the parallel scheduling algorithm schedules several nodes simultaneously. The resultant schedule may not be the same as the one produced by a sequential scheduling algorithm. Also, the schedule length will probably be longer than that produced by a sequential algorithm. Therefore, the quality of the schedule and the degree of the parallelism are normally in conflict. That is why we introduce a new parallel scheduling scheme, the hybrid scheme, in this thesis. The hybrid scheme combines the horizontal scheme and vertical scheme and takes advantages of both schemes to produce schedules with good speedup and acceptable schedule length.

Partitioning a scheduling problem into sub-problems and combining sub-results into final schedules are key issues in parallel scheduling algorithms. Several partition methods are studied. They are breadth-first-order partition, ALAP-order partition, and successor-number-order partition. After we schedule sub-problems, we get several sub-results. How to combine these sub-results into a final schedule with high scheduling quality remains an essential question. Several combination methods are presented. The simplest one is simple combination without permutation. The second method is to combine the sub-results according to the number of critical path nodes. The third method is to combine the sub-result by the order of actual running time. Therefore, using the same partition and different combination methods we can get different scheduling results with different time completions and scheduling qualities. We study different combinations of partition methods and sub-results combination methods under different conditions, such as the number of processing units or range of communication costs in order to find the

best parallel scheduling algorithms. A new parallel scheduling scheme is introduced in this thesis. We call it hybrid scheme because it utilizes both the horizontal scheme and the vertical scheme and benefits from the advantages of these two schemes. The hybrid scheme has architecture similar to the horizontal scheme. The difference is that in the horizontal scheme we use sequential scheduling algorithm to schedule the sub-graphs, but in the hybrid scheme, the vertical algorithm is used to schedule the sub-graphs.

We evaluate the parallel scheduling algorithms by efficiency and the scheduling quality. To find good parallel scheduling algorithms, we compare their completion time with sequential scheduling algorithms. We compare the speed-ups of different parallel scheduling algorithms. Scheduling quality is another import measurement. The goal is to find efficient parallel scheduling algorithms with high scheduling quality.

1.5 Previous Work

Little work has been done in designing parallel scheduling algorithms. In [5], Ahmad and Kwok studied a parallel scheduling algorithm called the parallel BSA (PBSA). The BSA algorithm is a sequential scheduling algorithm. The PBSA algorithm parallelizes the BSA algorithm in the horizontal scheme. Another research studied by Garey, Tarjan, and Yannakak [12], looks at the parallel scheduling algorithms. It presents two parallel scheduling schemes, one is a horizontal scheme, and the other is a vertical scheme. For the vertical scheme, the parallel scheduling algorithm called VPMCP algorithm is studied. MCP (Modified Critical-Path) algorithm is a sequential scheduling algorithm.

1.6 Thesis Organization

This thesis is organized as follows. In Chapter 2, the scheduling problem is formally defined, some related knowledge about scheduling is introduced, and the terminology used in this thesis is covered. Chapter 3 surveys previous research on scheduling algorithms and parallel static scheduling. In chapter 4, we study some designing issues of parallel scheduling. Several partition and combination algorithms are also presented. In particular, a new hybrid scheme that combines both the vertical scheme and the horizontal scheme is presented. In Chapter 5, experiment results and analysis are described. The experiment results of different combinations are described and compared, and the time complexity of parallel scheduling algorithms is analyzed. Finally, Chapter 6 outlines the conclusions reached based on the thesis experiments and looks at possible future research opportunities related to parallel scheduling algorithms.

Chapter 2

Terminology and Problem Definition

2.1 Scheduling

2.1.1 Task Scheduling Model

The main goal of scheduling algorithm is to find efficient ways to assign a set of cooperative tasks into a number of processors in the parallel processors system. A scheduling algorithm takes a task graph and a number of available processors as input, and produces an allocation and execution order of tasks to be processed by each processor as output. The task graph represents the problem need to be solved. The target machine contains the processors to which tasks are assigned, and a Gantt Chart can illustrate the schedule generated from a scheduling algorithm. Minimizing completion time and making the degree of parallelism are two major performance criteria.

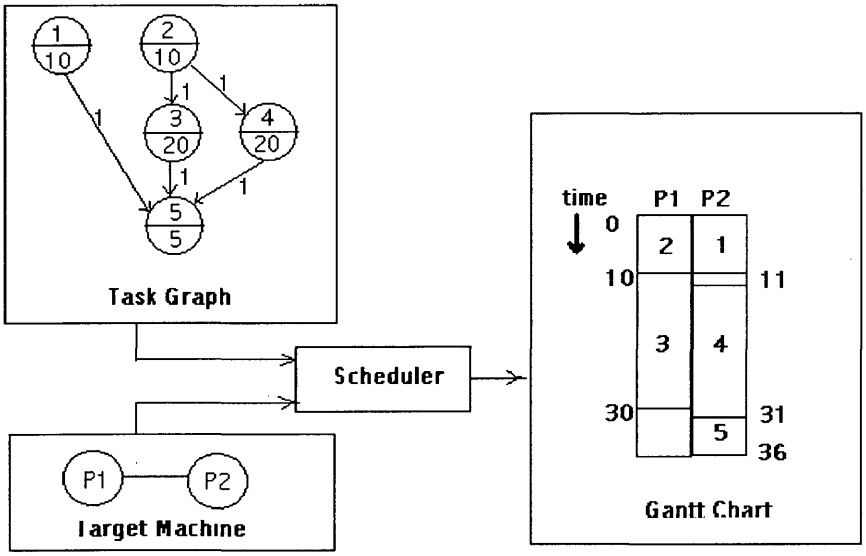


Figure 2.1 Task Scheduler Model

The Task Graph

The task graph is a simple but efficient way for representing the scheduling. A scheduling problem consists of separate cooperating and communication parts called tasks. Tasks communicate with one another. The time to communicate is measured as a communication time delay, and the time to execute a task is referred to task processing time. In Figure 2.1, circles represent tasks and relations are represented by the edges among circles. In each circle, the number in upper part is the task number. The numbers in the lower part of the circles are processing time for tasks. The numbers beside edges represents communication delays. The length of part is the summation of the weights of all nodes along the path including the initial and final node. For example, in Figure 2.1, the path length of the path (n2, n3, n5) is 37. The level of a node in a task graph is defined as the length of the longest path from the node to an exit node. An exit node is a one without successors. In Figure 2.1, node n5 is an exit node and the level of node n2 is 37.

Throughout this thesis, we follow a model with the following assumptions.

(1) *Acyclic Graph*: The task graph is acyclic

(2) *Static Topology:* The task graph is defined beforehand, and remains unchanged during program execution.

(3) *Non-homogeneous*: Task processing time and communication time delay need not be identical.

(4) *Non-preemption*: Once a task begins executing, it executes to completion.

(5) *Static Scheduling*: Once a task has been scheduled on a processor, the schedule is not changed.

Target Machine

The target machine is assumed to be made up of many heterogeneous processing elements connected using an arbitrary interconnection network. Each processing element can run one task at a time and all tasks can be processed by any processing element. We assume the following properties of the target machine:

(1) *Connectivity*: All processors are connected so that any processor can communicate with any other processor. For simplicity the queuing delay that may occur due to the congestion on the interconnection network is not considered.

(2) *Locality*: Transmission time between tasks located on the same processor is assumed to be zero time unit.

(3) *Identicality*: All processors are the same (speed, function, etc.)

(4) *Co-Processor I/O*: A processor can execute tasks and communicate with another processor at the same time.

(5) *Single Application*: Only one program is executed at a time on the parallel processing system.

Gantt Charts

A Gantt chart is used to represent the resulting schedule. A schedule is a two-dimension mapping of time and processing units. For each task, we need to know which processing unit will run the task as well as the beginning and finishing time of the task. The Gantt

chart gives an informal notion of the schedule where the start and finish time for all tasks can be easily shown. In a Gantt chart, the Y dimension represents the execution time and X dimension shows all available processors in target machine. For example in Figure 2.1 task 1 starts at time 0 and finishes at time 10, while task 4 starts at time 11 and finished at time 31. The shaded area between time 10 and 11 on processor P2 shows the communication delay as a result of message sent from task 2 on P1 to task 4 on P2.

2.1.2 Main Results

Based on the above model, finding an optimal schedule is a NP-complete problem. Different approaches have been suggested to find solutions of scheduling problems for real applications. Since we know optimal algorithms can only be found for special cases of scheduling problems, many heuristics algorithms have been created and presented.

The time complexity of a scheduling algorithm is a function of the task number and the processor number. As we know finding optimal schedule for scheduling problem is a NP-complete problem in its general form and in several restricted cases, even without considering communication cost among tasks. The following are formal definitions of some versions of the scheduling problem proven to be NP-complete [7]:

P1. *General Scheduling Problem*

Given a set T of N tasks, a partial order $<$ on T, weight Ai, $1 \leq i \leq n$, and m processors, and a time limit k, does exist a total function h from T to $\{0,1,\ldots k-1\}$ such that:

i) if $i < j$, then $h(i) + Ai \leq h(j)$

ii) for each I in T, $h(i) + Ai \leq k$

iii) for each t, $0 \leq t < k$, there are at most m values of i for which $h(i) \leq t < h(i)+Ai$?

The following problems are special cases of P1.

P2. *Single Execution Time Scheduling*

We restrict P1 by requiring $Ai = 1$, $1 \leq i \leq n$. (all tasks require one time unit)

P3. *Two Processor, on or two time-units scheduling*

We restrict P1 by requiring m = 2, and Ai in {1,2}, $1 \leq i \leq n$. (all tasks require one or two time units, and there are only two processors)

P4. *Two Processor, interval-order scheduling*

We restrict P1 by requiring the partial order < to be an interval order and m = 2.

P5. *Single Execution Time, Opposing Forests*

We strict P1 by requiring Ai = 1, $1 \leq i \leq n$, and the partial order < to be an opposing forest.

Problem P1 was proven to be NP-complete by Richard M. Karp in 1972. Problem P2 and P3 were proven to be NP-Complete b Jeffrey D. Ullman in 1975 [27]. Problem P4 was proven to be NP-complete by Christos H. Papadimitriou and Mihalis Yannakakis in 1979 [22]. Michael R.Garey, David S. Johnson, Robert E. Tarjan, and Mihalis Yannakakis in 1983 proved that P5 is also NP-complete [12].

Optimal Algorithms for Special Cases

Researchers still found some special cases of scheduling problem, which have polynomial time complexity. Polynomial algorithms can be obtained in the following cases: (1) when the task graph is a tree [15], (2) when the task graph is an interval order [11], and (3) when there are only two processors available [22]. In these three cases all tasks are assumed to have the same execution time and communication between tasks is not considered. The first case was introduced by Hu, who presented an algorithm, called the level algorithm, which can be used to solve the scheduling problem in linear time when the task graph is either an in-forest or an out-forest. The in-forest means each task has at most one immediate successor and the out-forest means that each task has at most one immediate predecessor [16]. In 1979, Papadimitriou and Yannakakis showed that interval ordered tasks can be scheduled in linear time on an arbitrary number of processors [22]. A task graph is an interval order when its elements can be mapped into interval on the real line and two elements are related if and only if the corresponding interval do not overlap. Finally, Coffman in 1976 proved that optimal algorithm exists when there are only two processors available [6].

Table 2.1 summarizes the time complexity of several version of scheduling problem when communication is not considered and the target machine is fully connected. In the table, n is the number of tasks and e is the number of edges in the task graph [7].

Task Graph	Task Execution Time	Number of Processors	Complexity
Tree	Identical	Arbitrary	O(n)
Interval order	Identical	Arbitrary	O(n)
Arbitrary	Identical	2	$O(e+n\alpha(n))$
Arbitrary	Identical	Arbitrary	NP-complete
Arbitrary	1 or 2 time units	≥ 2	NP-complete
Opposing forest	Identical	Arbitrary	NP-complete
Interval order	Arbitrary	≥ 2	NP-complete
Arbitrary	Arbitrary	Arbitrary	NP-complete

Table 2.1 Time Complexity Comparison of Scheduling Problem

Heuristics Algorithms

Because of the intensive computational complexity of finding optimal schedules, a need has arisen for a simplified sub-optimal approach for scheduling problem. Recent research in this area emphasized heuristic approaches. One class of scheduling heuristics, which was found to be simple and effective, is list scheduling. In list scheduling each task is assigned a priority, then a list of nodes is constructed in decreasing priority order. Whenever a processor is available, a ready node with the highest priority is selected from the list and assigned to the processor. If more than one node has the same priority, a node is selected randomly. The schedulers in this class differ only in the way that each scheduler assigns priorities to the nodes. Priority assignment results in different schedules because tasks are selected in a different order. The comparison among different node priorities (level, co-level and random) has been studied by Adam [1]. The comparison suggests that the use of level number of a node as its priority produces better result overall. Algorithm 2.1 presents a generic procedure of list scheduling [7].

Algorithm 2.1

1. Each node in the task graph is assigned a priority. A priority queue is initialized for ready tasks by inserting every task that has no immediate predecessors. Tasks are sorted in decreasing order of task priorities.

2. As long as the priority queue is not empty do the following :
   i) Obtain a task from the front of the queue
   ii) Select an idle processor to run the task
   iii) When all the immediate predecessors of a particular task are executed, that successor is ready to be inserted into the priority queue.

Communication Delay Versus Parallelism

When there is no communication delay between tasks, all ready tasks can be allocated to all available processors so that the overall execution time of the task graph is reduced. This situation may occur in a shared memory parallel processor where messages are passed at memory-cycle speeds. In fact, this is the basis of earlier schedulers that do not consider communication delay. If non-zero communication delay between tasks is considered, scheduling must be based on both the communication delay and time when each processor is ready for execution. It is possible that ready tasks with long communication delays could be assigned to the same processor as their immediate predecessors. For example, in Figure 2.2 the task graph has a communication delay Dx between task 1 and task 3. If Dx is bigger than execution time of task 2 as shown on Gantt Chart A, it would be better to assign task 3 to P1, which processed its immediate predecessor. Conversely, if the communication delay Dx is less than the execution time of task 2, it would be better if task 3 is assigned to P2 instead. Gantt Chart B illustrates this situation. Hence, adding communication delay constraint increases the difficulties of reaching an optimal schedule because the scheduler have to check the start time of each node on each available processor to select the best one. It is not always preferable to increase the degree of parallelism by simply starting each task as soon as possible. Distributing parallel tasks to as many processors as possible tends to increase the

communication delay, which contributes to the overall execution time. In short, there is a trade-off between taking advantage of maximal parallelism versus minimizing communication delay. This problem is usually referred to as the max-min problem for parallel processing.

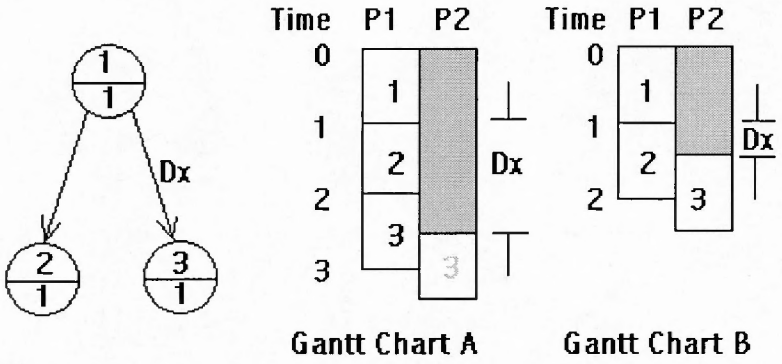


Figure 2.2 The Scheduling consideration due to communication delay

General List Heuristic with Communication

If we consider communication cost in list heuristic, the algorithm would be a little more complex. Algorithm 2.2 shows the general list heuristic with communication delay [1].

Algorithm 2.2

Input :

(1) TG, Task Graph

(2) M, number of processors in the parallel system

Output

P_GC, Gantt chart consisting of an array [1..m] for each processor in the system (list of tasks ordered by their execution time on a processor, including task start time and finish time)

Begin

1) Level_graph (TG);
   {Calculate level number for each task in TG};
2) Init_Gantt (P_GC);
   {Reset Gantt chart of each processor to empty};

3) Init_R_queue (RQ);
{Insert all tasks having no immediate predecessors into the RQ, ready queue, in order by their level number };
4) Get_task (AN, RQ);
{Get AN, assigned task, from the front of the ready queue};
5) Assign_task (AN, O, P+GC, 1, RQ);
{Assign AN, thc assigned task, to processor 1 at start time 0};

repeat

6) Update_R_queue (RQ, AN, TG);
{Update the ready queue using the assigned task};
7) if RQ, ready queue, is not empty then
    7.1) Get_task (AN, RQ);
    {Get a new AN from front of RQ};
    7.2) Locate_P (AN, P_GC, P_L, ST);
    {Return P_L that has the smallest ST, AN's start time};
    7.3) Assign_task (AN, ST, P_GC, P_L, RQ);
    {Assign AN to processor P_L at start time ST};

Until all tasks in TG are assigned

End

2.2 Parallel Scheduling

To distinguish the processing elements (PE) that execute a parallel scheduling algorithm from the target PE, to which the macro data flow graph is to be scheduled, we call the executing PE the physical processing elements (PPE) and call the target processing element TPE. So the TPE is the target machine to which tasks are scheduled and the PPE refers to actual processor that runs the scheduling algorithm. In order to parallelize the scheduling process itself, we need to identify a set of nodes that can be scheduled in parallel, instead of identifying one node to be scheduled each time. There are two basic methods to parallelize scheduling algorithm. The first one is a strait-forward idea that can be described as follows. First, sort the nodes in a topological order and divide them into

several partitions, then each partition is scheduled by sequential scheduling algorithm simultaneously, and finally combine all sub-results into one final output. The second way of parallelization is trying to schedule more than one node simultaneously each time. Instead of one processor calculate the start time of one node in one TPE in a sequential schedule algorithm, we can calculate the start time of for different TPEs simultaneously using multiprocessors and schedule them to TPEs. We call these two approaches horizontal scheme and vertical scheme illustrated in Figure 1 in Chapter 1.

2.2.1 Horizontal Scheme

In sequential scheduling algorithm, only one PPE is supposed to schedule all nodes one by one. However, if we can divide all nodes into several partitions and guarantee that all precedence constraints are scheduled before any node in a succeeding partition, we can schedule different partitions at the same time in different PPEs and still get reasonable result. Sorting all nodes in a topological order guarantees the right nodes scheduling order among partitions. So we can outline the horizontal parallel scheduling algorithm as follow:

(1) Sort nodes using some topological order and then partition nodes into N equal blocks to be assigned to the N PPEs.

(2) Each PPE applies a sequential scheduling algorithm to its partition to produces a sub-schedule.

(3) Merge all sub-schedules into one final schedule.

Three topological sorting algorithms are explored in this thesis: breadth-first sort, ALAP sort and successor-number sort. All these three algorithms make sure that every node is scheduled after all of its predecessors. Breadth-first sort uses breath-first search order to sort the nodes. The as late as possible time (ALAP) of a node v is defined *as* $T_L(v) = T_{critical} - level(v)$, where $T_{critical}$ is the length of critical path in the task graph, and $level(v)$ is the length of the longest path from node v to an exit node, including node v. The ALAP

sort calculates and sorts the ALAP time of the nodes. The successor-number sort uses the orders of the successor number of the nodes.

One major issue in the horizontal scheme is how to resolve the dependence relationship among partitions. When a PPE schedule its partition, it may need information of other sub-schedules. Unfortunatcly, such information does not exist before these partitions are scheduled. We have to use estimated information instead. This information can help a node to determine its earliest time in the later partition.

In the second step, each PPE schedules its own partition and produces a sub-schedule. Then, all sub-schedules are merged into one final schedule. Since accurate information of all sub-schedules is not available at each PPE, it is very difficult to find the optimal permutation of TPEs between adjacent sub-schedules. Heuristic technique needs to be developed to determine how the merge process is completed.

We use three permutation methods. The first one is just simply concatenate the corresponding TPEs in different sub-schedules. The second method is a heuristic algorithm. First, TPEs are sorted by the number of critical-path nodes. Then, we merge the TPEs by this sort order. In this way, we schedule as many critical path nodes as possible to the same TPE, and the critical path length could be reduced. This means that the length of final schedule would be reduced. The third method is similar to the second one. The difference is that instead of using the number of critical path nodes as the sort order, this method uses the sum of the actual execution time of different TPEs.

Finally, we walk through the entire merged schedule to determine the actual start time of each node. Some refinement can be performed in this step. The succeeding PPE is not able to insert nodes to its preceding sub-schedule due to lack of information. This leads to some performance loss. It can be partially corrected at the merging time by inserting the nodes of a succeeding sub-schedule into its proceeding sub-schedule. We call it post-insertion. However, post-insertion is very time consuming and the improvement is not significant [26]. Thus, we do not use post insertion in this thesis.

2.2.2 Vertical Scheme

As discussed in the previous section, the start time of a node in different TPEs can be calculated simultaneously instead of calculating start time in TPEs one by one. This concept can be exploited to parallelize the scheduling algorithm.

Two methods can be used to exploit the parallelism in the vertical scheme. The simple way is that each time we calculate the start times of one node on different TPEs simultaneously and then assign it to the earliest time slot available. This method produces the same schedule as the sequential algorithm. However, since each time only one node is scheduled, the degree of parallelism is limited. The second method constructs a waiting list of nodes. Each time a number of nodes from the beginning of the list are scheduled simultaneously. Since each node obtains its earliest start time independently, a node may get its earliest start time that is earlier than one of its parent nodes. Or, two nodes may compete for the same time slot in a TPE. To solve the conflict problem, we may allow a conflicting node to be scheduled to its sub-optimal place. Therefore, if a node is found to be in conflict with its parent nodes or compete the same time slot with another node, it can be scheduled to its second best place, and so on, until it is scheduled to a non-conflict place. With this strategy, p nodes can be scheduled each time, where p is the number of TPEs. Conflict results in the degradation of schedule quality. However, since each time

several nodes are scheduled, the degree of parallelism is increased. So, there is a trade-off between the schedule quality and parallelism of parallel scheduling algorithm.

Chapter 3
Previous Work

Although the scheduling problem has been fully studied for several decades, little recent work has been done on parallel scheduling algorithms. The first attempt in designing a parallel algorithm for scheduling is made by Ishfaq Ahmad and Yu-Kwong Kwok in 1995 [2]. They provided a horizontal scheme algorithm that is called parallel BSA algorithm (PBSA). Another study in this area came from Min-You Wu [26], who presented horizontal and vertical schemes for parallel scheduling and provided two algorithms discussed in this section in details.

3.1 Parallel BSA Algorithm

3.1.1 BSA Scheduling Algorithm

The BSA algorithm is a low complexity static scheduling allocation algorithm for message-passing architectures. This algorithm considers factors such as communication delays, link contention, message routing and network topology. A Process Element (PE) list is constructed in a breadth-first order and the PE with the highest degree is selected as the pivot PE. This algorithm constructs a schedule incrementally by first injecting all the nodes to the pivot PE. At this point, it tries to improve the start time of each node by migrating it to the adjacent PEs of the pivot PE, only if the migration will improve the start time of the node. After a node is migrated to another PE, its successors are also moved with it. Then, the next PE in the PE list is selected to be the new pivot PE. This process is repeated until all the PEs in the PE list have been considered. The complexity of the BSA algorithm is $O(p^2en)$, where p is the number of TPEs, n the number of nodes, and e the number of edges in a graph.

3.1.2 Parallel BSA Algorithm

The PBSA algorithm parallelizes the BSA algorithm in the horizontal scheme. The nodes in the graph are sorted in a topological order and partitioned into P equal sized blocks. Each partition of the graph is then scheduled to the target system independently. The PBSA algorithm resolves the dependencies between the nodes of partitions by calculating an estimated start time of each parent node belonging to another partition called the remote parent node (RPN). This time is estimated to be between the earliest possible start time and the latest possible start time. After all the partitions are scheduled, the independently developed schedules are concatenated. The complexity of the PBSA algorithm is $O(p^2en/P^2)$, where P is the number of PPEs.

Estimated Start Time

In order to enable all the nodes in different partitions, and in order to know the finish time of their RPNs, a global information exchange among the PPEs is required. However, this can generate an excessive amount of communication overhead. So, in PBSA algorithms, an estimated start time of RPNs is used. These estimates are calculated using the following parameters.

Definition 1:

The earliest possible start time (EPST) of a node u is the largest sum of computation costs from an entry node to the node u but not including the node itself.

Definition 2:

The latest possible start time (LPST) of a node u is the sum of computation costs from the first node in the serial injection ordering to the node u but not including the node itself.

Definition 3:

The estimated start time (EST) of an RPN is given by

$$\alpha EPST + (1-\alpha)LPST,$$

where α is the importance factor and is equal to 1 if the RPN is a critical path node. Otherwise, it is equal to the length of the longest path from an entry node, through the RPN, to an exit node, divided by the length of the critical path. Here, the length of a path is the sum of the computation and communication cost along the path.

PBSA Algorithm

The PBSA algorithm is written in a host-node programming style. The host is responsible for all pre-scheduling and post-scheduling housekeeping work. This includes the serial injection process, the task graph partitioning process, the concatenation of partial schedules, and the resolution of any conflicts in partial schedules. All of the parallel PPEs concurrently schedule the partitions of the task graph assigned to them.

PBSA_Host( ):

(1) Load processor network topology and input task graph.

(2) Serial_injeciton( ).

(3) Partition the task graph into equal sized sets according to the number of PPEs available. Determine the ESTs and estimated TPEs for every RPNs in all partitions.

(4) Broadcast the processor network topology to all PBSA_Node( ).

(5) Send the particular graph partition together with the corresponding ESTs and estimated TPEs to each PBSA_Node( ).

(6) Wait until all PBSA_Node( ) finish.

(7) Concat_Schedules( ).

PBSA_Node( ):

(1) Receive the target processor network from PBSA_Host( ).

(2) Receive graph partition together with the RPN's information (i.e. estimated start times and TPEs) from PBSA_Host( ).

(3) Apply the serial BSA algorithm to the graph partition. For every RPN, its EST and estimated TPE are used for determining the earliest possible data available time of a node to be scheduled in the local partition.

(4) Send the resulting sub-schedule to PBSA_Host( ).

Concat_Schedules( ) :

For every pair of adjacent sub-schedules, apply following steps to them:

(1) Determine the earliest node in the later sub-schedule. Call this the leader node. Call its TPE the leader TPE.

(2) Concatenate all nodes, which are scheduled on the same TPE as the leader node, to a TPE in the former sub-schedule so that the leader node can start as early as possible.

(3) Concatenate the node on all other TPEs to the TPEs of the former sub-schedule in a breadth-first order, beginning from the neighbors of the leader TPE.

(4) Re-schedule the exit nodes in the latter sub-schedule so that they can start as early as possible.

(5) Walk through the whole concatenated schedule to resolve any conflict between the actual start times and the estimated start times.

3.2 VPMCP and HPMCP Algorithms

In [12], Min-You studies the parallelization of static scheduling and presents two parallel scheduling algorithms: VPMCP and HPMCP. Wu gives two schemes for the parallelization of scheduling algorithms, a horizontal scheme and a vertical scheme (see

Figure 1). The VPMCP algorithm is a vertical scheme parallel scheduling algorithm and the HPMCP is a horizontal scheme parallel scheduling algorithm. Both the VPMCP and the HPMCP algorithms are based on the serial scheduling algorithm MCP.

3.2.1 MCP Algorithm

The Modified Critical-Path (MCP) algorithm is a simple algorithm. It modifies the original critical path algorithm by incorporating the edge weights. It schedules a macro dataflow graph on a bounded number of PEs. To explain the MCP algorithm, we first define the ALAP (As-Late-As-Possible) time of a node. The ALAP time is defined as $T_L(n_v) = T_{critical} - level(n_v)$, where $T_{critical}$ is the length of the critical path, and $level(n_v)$ is the length of the longest path from node n_v to an exit node, including n_v. The path length includes both the computation time and the communication time. This means that the path length is the summation of the node weights and edge weights along the path. The ALAP time can be obtained by the ALAP binding, which is created by moving downward through the macro data flow graph. In an ALAP binding, each node is assigned the latest possible execution time that does not delay the execution of any node on the critical path.

The MCP Algorithm

1. Calculate the ALAP time of each node.
2. Sort the node list in an increasing ALAP order. Ties are broken randomly.

3. Schedule the first node in the list to the PE that allows the earliest start time, considering idle time slots. Delete the node from the list and repeat Step 3 until the list is empty.

In step 3, when determining the start time, idle time slots created by communication delays are also considered. Also, a node can be inserted into the first feasible idle time. This method is called an insertion algorithm. The complexity of MCP algorithms is $O(n^2)$, where n is the number of nodes in a graph.

3.2.2 VPMCP Algorithm

Before the VPMCP algorithm, a simple parallel version of the MCP algorithm, called VPMCP1, was studied. This version schedules one node each time so that it produces the same schedule as the sequential MCP algorithm. The schedule of each node is broadcasted to all PPEs, along with its parent information including the scheduled TPE number and time. Then, the start times of each node on different TPEs can be calculated in parallel. The node is scheduled to the TPE that allows the earliest start time. Consequently, if a PPE has any node that is a child of the newly scheduled node, the corresponding parent information of the node is updated.

The VPMCP1 algorithm parallelizes the MCP algorithm directly. It produces exactly the same schedule as MCP. However, since only one node is scheduled each time, the degree of parallelism is limited and the granularity is too fine. In VPMCP, a number of nodes are scheduled simultaneously to increase the granularity and to reduce communication cost.

However, nodes my conflict with each other when they are scheduled simultaneously and conflict may result in a degradation of the scheduling quality. To solve the conflict problem, the VPMCP algorithm allows a conflict node to be scheduled to its sub-optimal place. Therefore, when a node is found to be in conflict with its former nodes, it will be scheduled to the next non-conflict place. With this strategy, p nodes can be scheduled each time, where p is the number of TPEs.

VPMCP Algorithm

1. **(a)** Compute the ALAP time of each node and sort the node list in an increasing ALAP order. Ties are broken randomly.

   (b) Divide the node list with cyclic partitioning into equal sized partitions and each partition is assigned to a PPE. Initialize an available list. A node is available if all of its parent nodes (if any), have been scheduled and all of the nodes that have a higher priority have been scheduled or are in the available list. Sort the list in an increasing ALAP order.

2 **(a)** The first p nodes in the available list are broadcast to all PPEs by using parallel concatenation operation, along with their parent information. If there are less than p nodes in the available list, broadcast the entire available list.

   (b) Each PPE calculates a start time for the node on each of its TPEs. These start times are made available to every PPE by parallel concatenation.

   (c) A node is scheduled to the TPE that allows the earliest start time. If more than one node competes for the same time slot in a TPE, the node with the smallest ALAP time gets the time slot. The node that does not get the time slot

is then scheduled to the time slot that allows the second earliest start time, and so on.

(d) The parent information is updated for the children of the scheduled node. Delete these nodes from the available list and update the available list. Repeat this step until the available list is empty.

The time for calculating of the ALAP time and sorting the nodes is O(e+elogn). The parallel scheduling step is of order $O(n^2/P)$. Therefore, the complexity of the VPMCP algorithm is $O(e+nlogn+n^2/P)$, where n is the number of nodes, e the number of edges, and P the number of PPEs. The communication cost is O(2n) for VPMCP1 and O(2n/p) for VPMCP, where p is the number of TPEs.

3.2.3 HPMCP Algorithm

This algorithm employs the horizontal scheme discussed earlier. It first partitions the graph into several partitions and assigns each partition to one PPE. Each PPE then schedules its partition using MCP algorithm. The precedence constraint among partitions is ignored so that each partition can be scheduled independently. A node is treated as an entry node in its partition if all of its parent nodes are remote. Nodes are scheduled in the order of the ALAP priorities. Each PPE schedules the node of its partition starting from its local time zero. Then, adjacent sub-schedules are merged to form the final schedule. Corresponding TPEs in different partitions are concatenated together and no post-insertion is performed. The final schedule is then walked through to determine the actual start time of each node and the final schedule length.

HPMCP Algorithm

1. **(a)** Compute the ALAP time of each node and sort the node list in an increasing ALAP order. Ties are broken randomly.

   (b) Partition the node list into equal sized blocks and each partition is assigned to a PPE.

2. Each PPE applies the MCP algorithm to its partition to produce a sub-schedule, ignoring the edges between a node and its remote parent nodes. Schedule the first node n_i in the list to the TPE that allows the earliest start time $t(n_i)$. Delete n_i from the list and repeat this scheduling procedure until the list is empty.

3. Concatenate each pair of adjacent sub-schedules. A TPE in the later sub-schedule is concatenated to the counterpart TPE in the former sub-schedule, with no post-insertion performed. Walk through the schedule to determine the actual start time of each node by

   $$t(n_i) = \max\{\ t(n_k)+w(n_k),\ \ t(n_j)+w(n_j)+w(e_{j,i}) \mid n_j \text{ is parent of } n_i\ \}$$

   where n_k is the node that was scheduled immediately before ni.

The time for calculating the ALAP and sorting the nodes is O(e+nlogn). The second step of the parallel scheduling is of $O(n^2/P^2)$, and the third step takes O(e) time. Therefore, the complexity of the HPMCP algorithm is $O(e+n\log n+ n^2/P^2)$, where n is the number of nodes and e the number of edges in a graph, p is the number of TPEs and P the number of PPEs.

Chapter 4

Implementation

In the previous chapters, we discussed the current state of research in the area of parallel scheduling algorithms. At this point, several questions can be raised as to how well parallel scheduling algorithms perform and to how its performance compare to that of sequential scheduling algorithms? How to choose a parallel scheduling algorithm? Is there a parallel scheduling algorithm that is better than others all the time, and, if not, how do you choose a parallel scheduling algorithm under different circumstances? To answer these questions, we need to study the time complexity of the algorithms and implement these algorithms and compare the obtained results. Here, rather than comparing the algorithms theoretically, we simulated these algorithms and compared the results. The objective of this thesis is to find the answers to the questions listed above, to compare the efficiency of the parallel scheduling algorithm and to find the most efficient criteria needed to select different parallel scheduling algorithms.

Three parallel schemes are simulated in this study. They are the horizontal scheme, the vertical scheme, and the hybrid scheme. In the horizontal scheme, the task graph is partitioned into sub-graphs according to the number of physical processing units. Then, each partition is simultaneously scheduled by a sequential scheduling algorithm and sub-schedules for these sub-graphs are created. Finally, these sub-schedules are concatenated together into a final schedule. Several partition and concatenation methods are used to produce many variant algorithms of this scheme. The second scheme is the vertical scheme, in which several tasks are simultaneously scheduled. If conflicts happen, tasks with lower priority will be scheduled to their sub-optimal time slots. The number of tasks scheduled each time determines the number of target processing units. The third scheme,

denoted by the hybrid scheme, combines the horizontal and vertical schemes and is introduced by this thesis. The architecture is similar to the horizontal scheme, but when the sub-graphs are scheduled, the hybrid scheme uses the vertical scheduling algorithm instead of the sequential scheduling algorithm. Therefore, the hybrid scheme benefits from both the horizontal and the vertical scheme.

This chapter is organized as follows. Section 4.1 discusses the simulation environment, the horizontal and vertical schemes are discussed in Sections 4.2 4.3, respectively. Finally, Section 4.4 introduces the concept of the hybrid scheme for parallel scheduling algorithms.

4.1 Simulated System

The simulator is written using C programming language. The compiler is Microsoft Visual C++ 5.0. The program is written following the ANSI C standard. Three separate programs implement the three schemes studied in this thesis, but some routines are common among them, such as the graph generator and some data structure related functions.

Graph Generator

The same graph generator is used for all schemes because different algorithms are comparable only when they have the same input task graphs. We consider the communication among processors, as well as the task processing time. The graph generator can either generate a new graph according to the parameters provided or save

the graph on a file, or it can just use an existing graph. Listed below are the parameters for the graph generator:

Node number

Edge number

Maximum node processing time

Maximum communication time

Communication to computation ratio (CCR)

The node number determines the size of the graph. The edge number reflects the number of dependencies among nodes, which determines the complexity of the graph. When the graph is generated, the CCR (communication to computation ratio) of the graph can specified.

Input Parameters

While data structures and implementation detail differ from one scheme to the other, the main structure and most of the input parameters are common. The input parameters that are common to all schemes include:

Number of physical processing elements (PPE)

Number of target processing elements (TPE)

Task graph

Processing time for all nodes

Number of nodes in a task graph

For the hybrid scheme, the program also needs to know how many PPEs are used in each horizontal partition.

Performance Measurements

Several performance measurements are used to compare different algorithms. The first one is the executing time of the algorithm. The speed of the algorithm is an essential criteria for a good algorithm. The second important thing is the length of the final schedule. Some algorithms have higher degree of parallelism than others do and some algorithms may produce schedules with better schedule quality. Depending on the circumstance and what is more important, these two measurements could have different priorities. Another way to compare the speed of the algorithm is to calculate the speed-up from the sequential algorithm.

4.2 Horizontal Scheme

There are several steps involved in horizontal scheme implementation. First, the graph is partitioned into several sub-graphs according to the number of the physical processing elements (PPE). Then, for each sub-graph, a general list schedule algorithm is applied and a sub-schedule is produced. The next step is to transfer the node numbers in the sub-schedule to the original node number in the input task graph. After that, a permutation

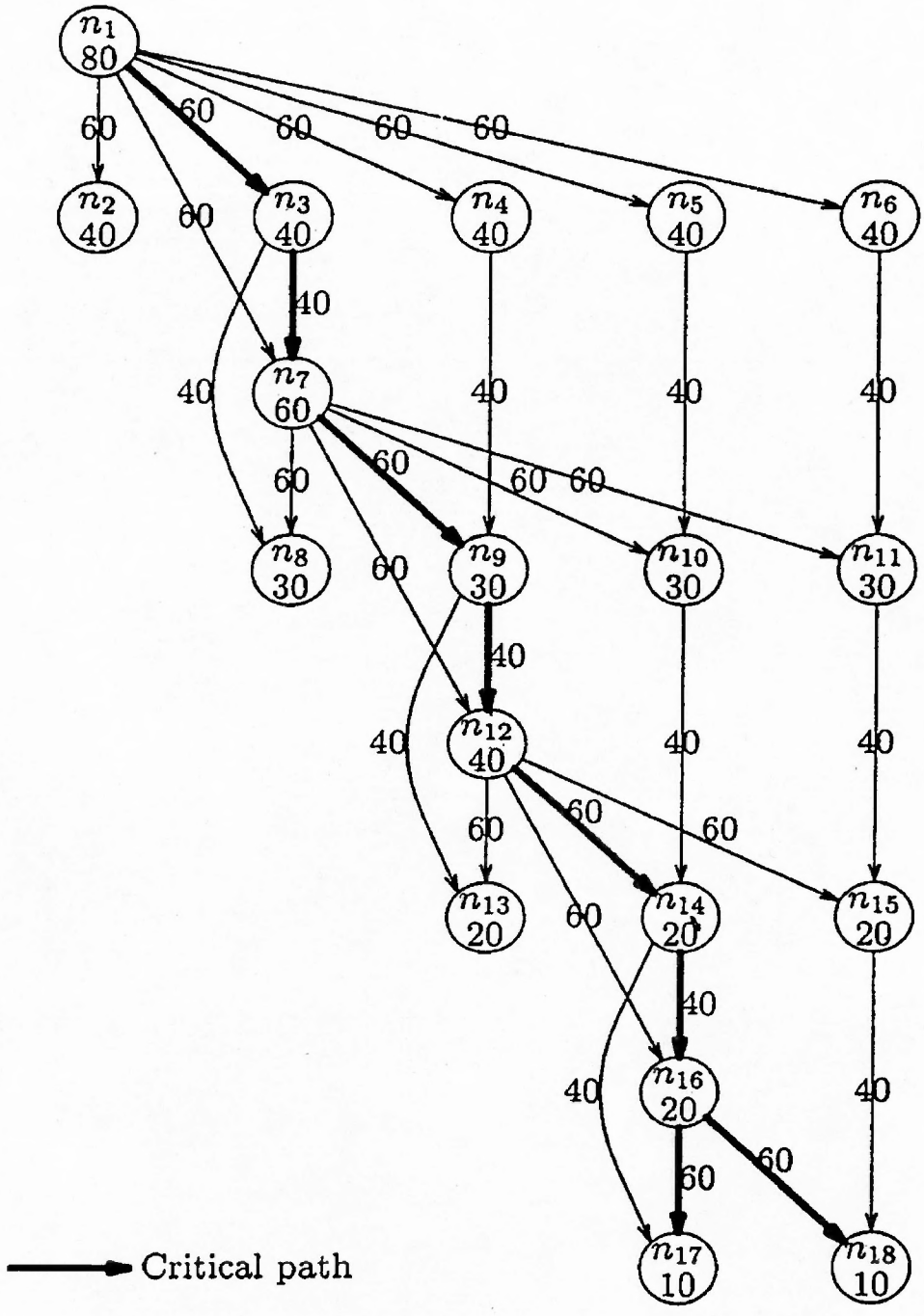


Figure 4.1 A task graph

method is used against the sub-schedules to prepare for the final combination. At last, all sub-schedules are combined into one final schedule.

4.2.1 Algorithm

Here is the horizontal scheme algorithm:

Horizontal Algorithm

Step 1: Generate graph and get some parameters

- Get tasks number, PPE, TPE, MAX_COM_COST, MAX_PRO_COST, CCR (Communication-to-computation ratio), edges number, partition method and combination method.
- Generate communication cost matrix with random number x, $0 < x <$ MAX_COM_COST and the number of non-zero value in the matrix is En.
- Generate task cost array with random number x, $0 < x <$ MAX_PRO_COST+1.

Step 2: Partition the graph

- One of the three partitions is used
  (1) Breadth-first-partition.
  (2) ALAP-partition.
  (3) Successor-number-partition.
- Sort each partition.

Step 3: According to the partition, divide the task graph into several sub-graphs and divide the processor cost list to some sub-lists.

Step 4: Schedule all sub-graphs using a general list scheduling algorithm, which produces a corresponding Gaunt Chart for each sub-graph (parallel action).

Step 5: Transfer node numbers in all sub-graphs Gaunt Charts back to the original node numbers in the beginning graph.

Step 6: Switch rows in the sub-graph Gaunt Chart according to the permutation method used to combine the sub-graph Gaunt Chart. There are three permutation methods and one of them is used:

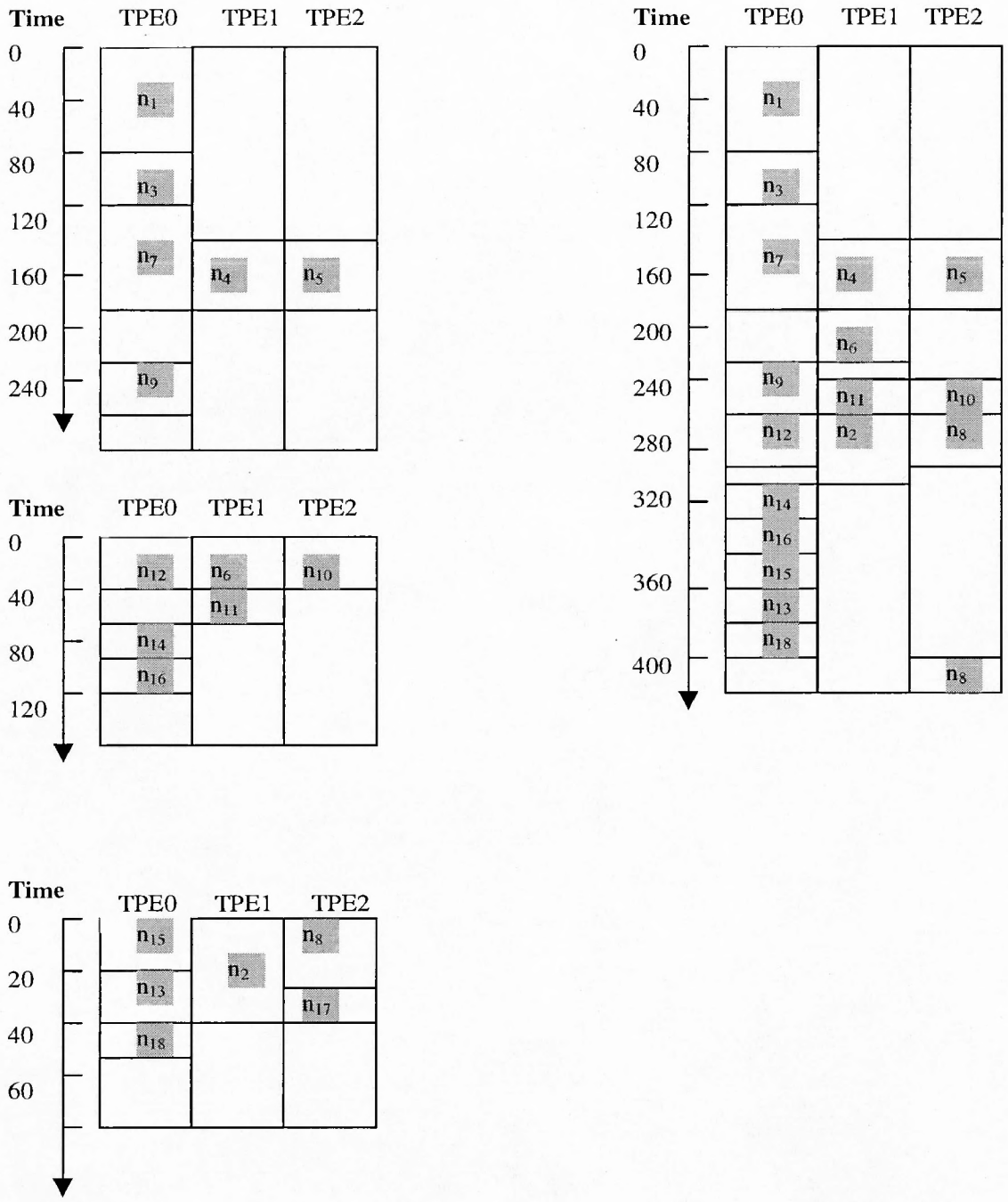
(1) Simply combine sub-graph Gaunt Chart with the corresponding rows.

(2) Order rows by the number of critical path nodes.

(3) Order rows by the actual execution time of each row.

Step 7: Combine all sub-graph Gaunt Charts into one final output.

Figure 4.2 shows a running example of the horizontal scheme algorithm, which schedules the task graph in Figure 4.1 to three TPEs. The graph contains three partitions, each has six nodes. Nodes n1, n3, n7, n4, n9 and n5 are assigned to PPE0, nodes n12, n6, n10, n14, n11 and n16 are assigned to PPE1, and nodes n15, n2, n8, n13, n17 and n18 are assigned to PPE2. Note that nodes n6, n10 and n12 in PPE1 and nodes n2, n8, n13, n15 and n17 in PPE2 become ready nodes when ignoring the dependence between partitions. Each partition of nodes is scheduled independently. The partial schedules are shown in Figure 4.2 (a). They are concatenated to form the final schedule without permutation. Furthermore, no post-insertion is performed. Finally, we walk through the entire schedule to determine the actual start time of each node, as shown in Figure 4.2 (b).



(a) (b)

Figure 4.2 Running example for the horizontal scheme

4.2.2 Partition and Permutation Methods

We implemented the horizontal scheme using different kinds of topological sorts and concatenation permutations. Thus, by using different combinations of topological sorts and permutations, different horizontal algorithms are created. Also, in order to speed up the algorithms, information estimation and post insertion are not used.

Breadth-first-partition

Breadth first partition is direct and the simplest way to divide the graph into sub-graphs, and, as a consequence, it is very fast and useful.

Step1: Initialize the queue with nodes that have no preceding node.

Step2: While the queue is not empty, repeat the following two sub-steps:

-Get first node X from the queue and store the node.

-Add all of X's children nodes to the tail of the queue.

ALAP-partition

The second way to partition is to use ALAP order of the graph. The partitions created by this method will lead to schedules with a short length, but this method is time consuming. In addition, because partition is a sequential part in the algorithm, this partition method will slow down the algorithm.

Step 1: Calculate critical path.

Step 2: Find the largest value MAX_LENGTH for critical path.

Step 3: Find nodes on critical path.

Step 4: Calculate level of nodes.

Step 5: Calculate ALAP order of nodes.

Successor-number-partition

The third partition method in this study is to partition the graph by the order of the successor numbers of the nodes. It is faster than ALAP-Partition, but slower than breadth-first partition.

Step 1: Initialize queue with nodes without preceding nodes.

Step 2: While queue is not empty, repeat the following sub-steps

-Get first node X from the queue.

-For each parent node Y of X,

for each parent node Z of Y,

initialize the dependence from Z to X .

- For each children node W of X,

add W to the end of queue.

Step 3: Calculate the successor number of each node.

Step 4: Sort the array that stores the number of successors for all nodes.

Order rows by the number of critical path nodes

By ordering rows by the number of critical path nodes, we are able to calculate the number of nodes that are in the critical path in each TPE and it sorts the TPEs by the number of critical path nodes.

Order rows by the actual execution time of each row

By ordering rows using the actual execution time of each row, the TPEs are sorted by their total execution time. Since there are communication delays between tasks, some tasks have to wait for tasks from different TPEs to finish. Thus, there are many idle time slots in each TPEs. The actual execution time is equal to the total processing time of all nodes in the TPE.

4.3 Vertical Scheme

There are two ways to exploit parallelism in a vertical scheme. The first approach is simple and straight froward. Instead of scheduling one node at a time in a sequential scheduling algorithm, we can calculate the start time of one node on different TPEs simultaneously. This way, the parallel scheduling algorithm produces the same schedule as the sequential one. However, the degree of parallelism is low because only one node is scheduled each time. The second way increases the degree of parallelism and reduces communication cost, but produces schedules with lower schedule quality. It schedules several nodes simultaneously each time. However, the start time of different nodes may conflict with each other and this would need to be taken care of. Since several nodes are scheduled at the same time, each node obtains its earliest start time independently. A conflict happens when a node get its earliest start time and it is earlier than one of its former nodes, or when two nodes compete for the same time slot in the same TPE. To solve the conflict problem, we allow a conflict node to be scheduled to its sub-optimal place. Therefore, if a node is found to be in conflict with its former nodes or it competes with the same time slot with another node, it can be scheduled to its second best place, and so on, until it is scheduled to a non-conflict place. Conflict may result in the degradation of schedule quality. Thus, there is a trade-off between the schedule quality and the parallelism of a parallel scheduling algorithm.

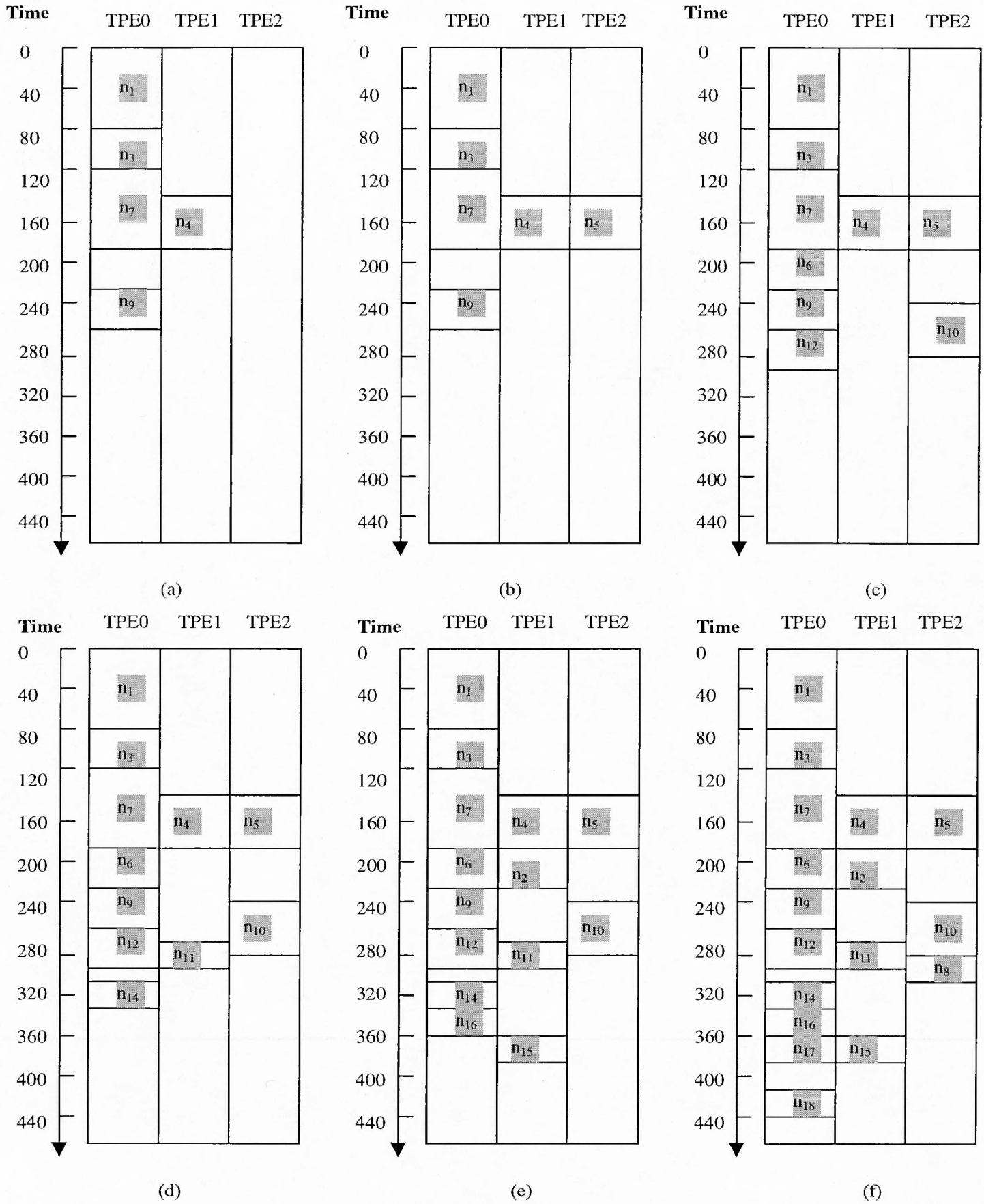
In the second approach, we simulate the vertical scheme. Each time, we scheduled p nodes simultaneously, where the p is equal to the number of TPEs. If a conflict occurred, the node with the highest priority got the time slot and the other were scheduled to their sub-optimal time slot, and so on. The vertical algorithm implemented is described below.

The Vertical Scheme Algorithm

1. Initialize an available list. A node is available if all of its parent nodes, if any, have been scheduled.
2. The first p nodes in the available list are broadcast to all PPEs, along with their parent information. If there are less than p nodes in the available list, broadcast all of them.
3. Each PPE calculates the start times of nodes on each of its TPEs.
4. A node is scheduled to the TPE that allows for the earliest start time. If more than one node competes for the same timeslot in a TPE, the node with the highest priority gets the time slot. The node that does not get the time slot is then scheduled to the time slot that allows the second earliest start time, and so on.
5. The parent information is updated for the children of scheduled nodes. Update the available list by adding available nodes.
6. Repeat step 2 to step 5 until the available list is empty.

Figure 4.3 shows a running example of the vertical scheme algorithm that schedules the task graph in Figure 4.1 to three TPEs. In the beginning, only node n1 is available. After it is scheduled to PE0, nodes n2, n3, n4, n5, and n6 are ready. However, nodes n2, n4, n5, and n6 have lower priorities than node n7, which is not ready. Thus, only node n3 is in an available list and is scheduled. At this time, nodes n2, n4, n5, n6, and n7 are ready, but only nodes n7 and n4 are in the available list because n2, n5, and n6 have lower priorities than node n9, which is not ready. Both nodes n7 and n4 have the earliest start time 120, at PE0, and, thus, they conflict with each other. Node n7 has a higher priority than node n4, hence, it is selected to be scheduled to the time slot. Node n4 is scheduled to its sub-optimal place in PE1. The partial schedule is shown in Figure 4.3 (a). Now, nodes n9 and

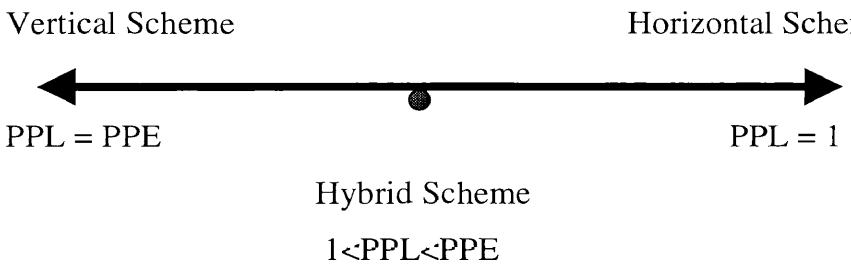
n5 can be scheduled. Node n9 is scheduled to PE0 and node n5 to PE2. This allows for the earliest start time for the two nodes, respectively. The partial schedule is shown in Figure 4.3 (b). Next, nodes n12, n6, and n10 can be scheduled. Nodes n12 and n6 are scheduled to PPE0, but in different time slots. Node n10 is scheduled to PE2. The partial schedule is shown in Figure 4.2 (c). Then, nodes n14 and n11 are scheduled to PE0 and PE1, respectively. The partial schedule is shown Figure 4.3 (d). At this point, nodes n16, n15, n2, n8, and n13 can be scheduled but only the first three nodes are scheduled since we only have three PEs. Node n15 conflicts with node n16 on PE0 and is scheduled to PE1. Figure 4.3 (e) shows the partial schedule. Then, nodes n8, n13, and n17 are scheduled. Finally, node n18 is scheduled and the final schedule is shown in Figure 4.3 (f).


Time
TPE0
TPE1
TPE2
0
40
80
120
160
200
240
280
320
360
400
440
n1
n3
n7
n4
n9
(a)
n5
(b)
n6
n10
n12
(c)
n14
n11
(d)
n2
n16
n15
(e)
n17
n18
n8
(f)

4.4 Hybrid Scheme

We know there is a trade-off between the schedule quality and the parallelism of a parallel scheduling algorithm. When the parallelism is fully exploited, the resulting schedule has a longer schedule length with lower quality. We can find this trade-off in the horizontal scheme and the vertical scheme. In the horizontal scheme, the schedule quality is pretty good, but the parallelism is not exploited efficiently. In the horizontal algorithms we studied in this thesis, only 10% to 18% of running time is used for the parallel part of the algorithms. There is a lot of overhead in horizontal algorithms, like partitioning the graph and combining the sub-schedules. On the other hand, the vertical scheme has better parallelism because there are only a few extra works need to be done before and after the parallel execution of the algorithm. The vertical algorithm has a better speedup over the sequential scheduling algorithm, however, the schedule it produces is not as good as that found in horizontal algorithms.

Is there a good way to achieve better parallelism while still getting acceptable schedule length? In this thesis, we try to solve this problem by introducing a new parallel scheduling scheme, the hybrid scheme. We call it a hybrid scheme because it utilizes both the horizontal scheme and the vertical scheme and it benefits from the advantages of these two schemes. In fact, we can say it is a type of variant of the horizontal scheme, since it has architecture similar to the horizontal scheme. The major difference lies in the fact that in the horizontal scheme we use a sequential scheduling algorithm to schedule the sub-graphs, but in the hybrid scheme, the vertical algorithm is used to schedule the sub-graphs. Another difference is the number of PPEs in each horizontal partition. In the horizontal scheme, each partition has only one PPE, but in the hybrid scheme, each partition may have more than one PPE. The number of PPEs , called PPL (PPEs Per Layer), in each partition may affect the performance of the hybrid scheme. The following figure shows the relationships among the horizontal scheme, the vertical scheme and the hybrid scheme:



The hybrid scheme has the flexibility to achieve good parallelism as well as an acceptable schedule quality. It combines the horizontal scheme and vertical scheme and takes advantage of these two schemes. Changing the value of PPL, we can get different schedules with different speedups. When the PPL is decreased to 1, the algorithm becomes a horizontal algorithm. On the other hand, when the PPL is equal to PPE, it becomes a vertical algorithm.

The following is the hybrid scheme algorithm:

Hybrid Scheme Algorithm

Step 1 : Generate graph and get some parameters.

Step 2 : Partition the graph.

Step 3 : According to partition, divide task graph into several sub-graphs and divide processor cost list to some sub-lists.

Step 4 : Schedule all sub-graphs using vertical parallel scheduling algorithms, which produces a corresponding Gaunt Chart for each sub-graph.

Step 5 : Transfer node numbers in all sub-graph Gaunt Charts back to the original node numbers in the beginning graph.

Step 6 : Switch the rows in the sub-graph Gaunt Charts according to the permutation method used to combine the sub-graph Gaunt Charts.

Step 7 : Combine all sub-graph Gaunt Charts into one final output.

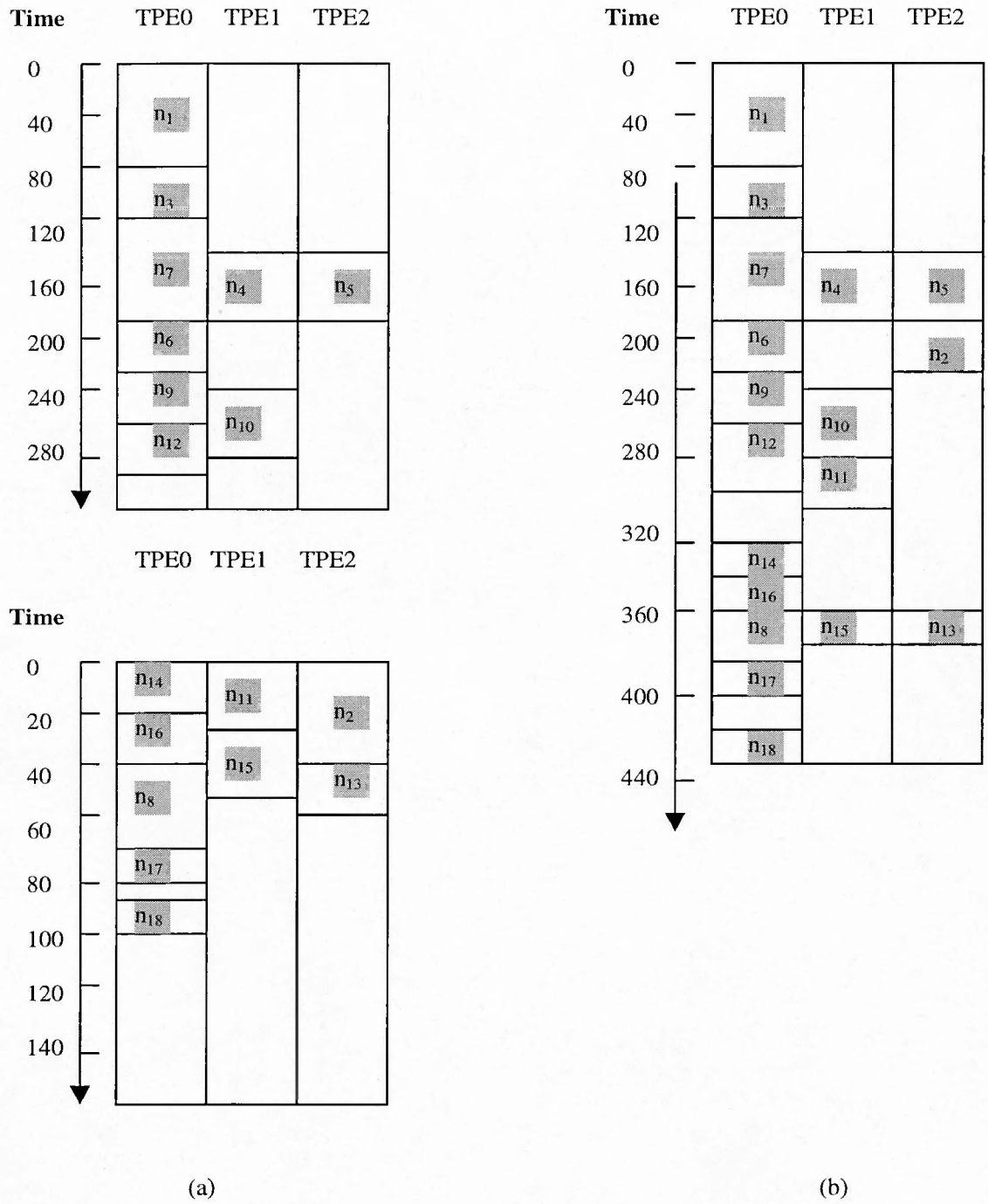


Figure 4.4 Running example for the hybrid scheme

Figure 4.4 shows a running example of the horizontal scheme algorithm that schedules the task graph in Figure 4.1 to three TPEs. First, the graph is partitioned into two partitions, each has nine nodes. Nodes n1, n3, n7, n4, n9, n5, n12, n6, and n10 are assigned to PPE0; nodes n14, n11, n16, n15, n2, n8, n13, n17, and n18 are assigned to PPE1. Note that nodes n2, n8, n11, n13, and n14, in PPE2, become ready nodes when ignoring the dependence between partitions. Each partition of nodes is scheduled independently using the vertical scheme algorithm. In each partition, there are two PPEs used to schedule the sub-graphs. The partial schedules, produced for the two partitions by the vertical algorithm, are shown in Figure 4.4 (a). They are concatenated to form the final schedule without permutation and no post-insertion is performed. At the end, we walk through the entire schedule to determine the actual start time of each node, as shown in Figure 4.4 (b).

Chapter 5

Result Analysis

In this chapter, we present the results of the simulating experiments of different parallel scheduling schemes. We implemented parallel scheduling algorithms using horizontal schemes, vertical schemes, and hybrid schemes. Sequential scheduling algorithm was also implemented for comparison purposes. Although many parameters can be used to study the performance of parallel scheduling algorithm, in this thesis, we only focus on two important parameters: the schedule length and the speedup; the schedule length represents the schedule quality and the speedup reflects the degree of the parallelism of the algorithm. Horizontal scheme algorithms produce efficient schedules, but it has a lot of overhead and the speedup is not good. The vertical scheme algorithms are faster with better speedup, but they produce schedules with longer schedule length. The hybrid scheme explores a way to deal with the trade off between the degree of parallelism and the schedule quality. It identifies a parameter, called PPL (PPEs per layer), to balance the tradeoff between parallelism and speedup. If the PPL is equal to 1, then the hybrid algorithm acts as if it is the horizontal algorithm. If the PPL is equal to PPE, then the hybrid algorithm would resemble the vertical algorithm.

5.1 Input and Output

Input – To conduct the simulation process, some inputs are needed. These inputs include:

- PPE - Number of physical processing element.

- TPE - Number of target processing element.
- G - Task graph.
- T(i) - Processing time for each node.
- Tn - Number of nodes in task graph.
- CCR - Computation to Communication Ratio.

Output - Two measurements are used to compare the scheduling algorithms. One is the schedule length and the other is the speedup of parallel algorithms. Thus, schedule length and running time are the outputs from the simulation program. A speedup is defined by S = Ts / Tp, where Ts is the execution time of a sequential algorithm (general list schedule algorithm) and Tp is the parallel algorithm execution time.

5.2 The Results of the Horizontal Scheme Based Algorithms

In this section, we report the results of implementing several parallel schedulers that employ the horizontal scheme. In Tables 5.1, 5.2, and 5.3, graphs with 1000 nodes and 3500 edges are scheduled using different horizontal scheme based algorithms. In Tables 5.1, 5.2, and 5.3, there are four TPEs. We use three partition methods and three merge techniques, as discussed in Chapter 4, in implementing the horizontal scheme, thus, resulting is a total of nine possible algorithms. We compare the schedules length, running time, and speedup of these algorithms with the sequential algorithm. The sequential algorithm used here is a general list scheduling algorithm instead of an optimal sequential algorithm. Although using the optimal sequential algorithm for comparison will produce

better speedups for the parallel scheduling algorithms, the heuristic algorithm gives more meaningful results.

The three partition methods are:

- Partition(1) : Breadth-first partition.
- Partition(2) : ALAP partition.
- Partition(3) : Partition by the order of number of successors.

The three combination (merge) methods are:

- Combine(1) : Simple combination without permutation.
- Combine(2) : Merge by the order of number of critical path nodes.
- Combine(3) : Merge by the order of actual execution time.

CCR	Number of PPEs	Partition(1) & Combine(1)			Partition(1) & Combine(2)			Partition(1) & Combine(3)		
		Time	Length	S	Time	Length	S	Time	Length	S
0.1	Seq	0.77	12825	-	0.77	12825	-	0.77	12825	-
	2	0.58	12834	1.33	0.61	12838	1.17	0.545	12835	1.41
	4	0.42	12812	1.83	0.45	12845	1.71	0.43	12931	1.79
	8	0.39	12843	1.97	0.43	12851	1.79	0.41	12981	1.88
1	Seq	0.78	1392	-	0.78	1392	-	0.78	1392	-
	2	0.695	1393	1.12	0.75	1394	1.04	0.605	1395	1.29
	4	0.35	1391	2.23	0.45	1393	1.73	0.495	1404	1.58
	8	0.41	1395	1.90	0.44	1392	1.77	0.443	1412	1.76
10	Seq	0.77	1444	-	0.77	1444	-	0.77	1444	-
	2	0.465	1458	1.66	0.57	1445	1.35	0.565	1458	1.36
	4	0.42	1519	1.83	0.45	1450	1.71	0.39	1524	1.97
	8	0.43	1560	1.79	0.50	1452	1.54	0.56	1567	1.38

Table 5.1 Horizontal algorithms comparison

CCR	Number of PPEs	Partition(2)& Combine(1)			Partition(2)& Combine(2)			Partition(2)& Combine(3)		
		Time	Length	S	Time	Length	S	Time	Length	S
0.1	Seq	0.77	12825	-	0.77	12825	-	0.77	12825	-
	2	0.815	12832	0.945	0.95	12846	0.81	0.84	12846	1.73
	4	0.70	12870	1.10	0.762	12881	1.01	0.802	13024	0.96
	8	0.683	12908	1.12	0.733	12835	1.05	0.783	13050	0.98
1	Seq	0.78	1392	-	0.78	1392	-	0.78	1392	-
	2	0.86	1407	0.91	0.88	1403	0.89	0.87	1405	0.90
	4	0.705	1395	1.11	0.78	1401	1.00	0.815	1405	0.96
	8	0.733	1415	1.06	0.74	1410	1.05	0.733	1426	1.06
10	Seq	0.77	1444	-	0.77	1444	-	0.77	1444	-
	2	0.885	1423	0.87	0.86	1425	0.90	0.765	1423	1.01
	4	0.825	1433	0.93	0.88	1430	0.88	0.875	1441	0.88
	8	0.903	1405	0.85	0.89	1428	0.87	0.67	1498	1.15

Table 5.2 Horizontal algorithms comparison

CCR	Number of PPEs	Partition(3)& Combine(1)			Partition(3)& Combine(2)			Partition(3)& Combine(3)		
		Time	Length	S	Time	Length	S	Time	Length	S
0.1	Seq	0.77	12825	-	0.77	12825	-	0.77	12825	-
	2	0.95	12863	0.81	0.96	12833	0.80	0.93	12854	0.83
	4	0.87	12834	0.89	0.98	12838	0.79	0.85	12891	0.91
	8	0.827	12943	0.93	0.97	12873	0.79	0.73	13171	1.05
1	Seq	0.78	1392	-	0.78	1392	-	0.78	1392	-
	2	0.965	1394	0.81	0.98	1393	0.80	1.04	1396	0.75
	4	0.947	1399	0.82	0.97	1396	0.80	0.92	1403	0.85
	8	1.01	1395	0.77	1.10	1399	0.71	1.053	1419	0.74
10	Seq	0.77	1444	-	0.77	1444	-	0.77	1444	-
	2	1.13	1415	0.68	1.23	1413	0.63	1.155	1415	0.67
	4	0.982	1430	0.78	1.12	1420	0.68	1.077	1427	0.71
	8	0.953	1459	0.81	0.97	1451	0.79	0.843	1510	0.91

Table 5.3 Horizontal algorithms comparison

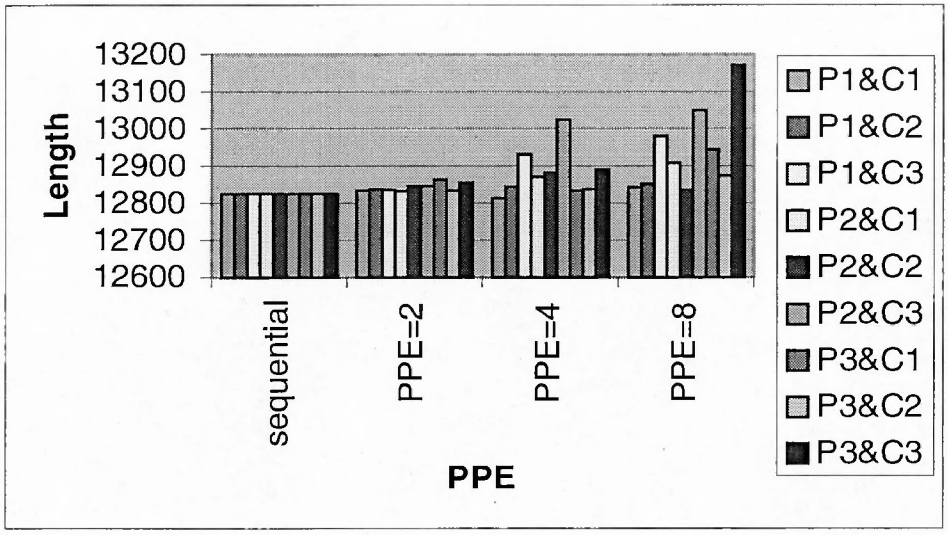


Figure 5.1 Comparison of schedule lengths of horizontal algorithms

Looking at Tables 5.1, 5.2, and 5.3, we can see that the schedule lengths produced by the horizontal algorithms are very close to the ones produced by sequential algorithms, as seen in Figure 5.1. However, the speedup is relatively low. The low speedup values are mainly due to a lot of sequential work before and after the simultaneous scheduling of different partitions. We find out that only 10% to 18% of the running time is for the parallel part of the algorithm. As a result, the algorithms in Table 5.1 are faster than the algorithms in Tables 5.2 and 5.3 due to the faster partition method.

5.3 Analysis Result for Vertical Scheme Algorithm

Note that in the vertical scheme, the number of PPEs cannot be larger than the number of TPEs because each TPE must be kept in a single PPE. The results shown in Tables 5.4, 5.5, and 5.6 list the schedule length, running time and speedup produced by the vertical algorithm under different CCRs (Communication to computation ratio). Graphs of 500,

1000, and 1500 nodes are scheduled. The numbers of edge in the graphs are four times more than the numbers of nodes and TPEs = 2*PPEs in the graphs.

Number of PPEs	Graph Size								
	500			1000			1500		
	Time	Length	S	Time	Length	S	Time	Length	S
Seq	0.16	6272	-	0.72	13032	-	1.31	19001	-
2	0.17	6630	0.94	0.517	13774	1.39	0.74	9141	1.77
4	0.104	3776	1.54	0.366	7179	1.97	0.48	4873	2.73
8	0.052	2289	3.08	0.213	4040	6.46	0.235	2701	5.57

Table 5.4 Vertical algorithm where CCR = 0.1 (10/100)

Number of PPEs	Graph Size								
	500			1000			1500		
	Time	Length	S	Time	Length	S	Time	Length	S
Seq	0.17	696	-	0.82	1401	-	1.32	2097	-
2	0.195	874	0.87	0.521	1684	1.57	1.755	2517	0.75
4	0.179	488	0.95	0.362	892	2.26	0.905	1300	1.46
8	0.059	356	2.89	0.227	509	3.61	0.552	719	2.39

Table 5.5 Vertical algorithm where CCR = 1 (10/10)

Number of PPEs	Graph Size								
	500			1000			1500		
	Time	Length	S	Time	Length	S	Time	Length	S
Seq	0.16	935	-	0.66	1497	-	1.32	2088	-
2	0.19	1701	0.84	0.314	971	2.1	1.79	2829	0.74
4	0.069	1502	2.32	0.196	757	3.37	0.833	2129	1.58
8	0.053	1400	3.02	0.095	729	6.9	0.599	1910	2.20

Table 5.6 Vertical algorithm where CCR = 10 (100/10)

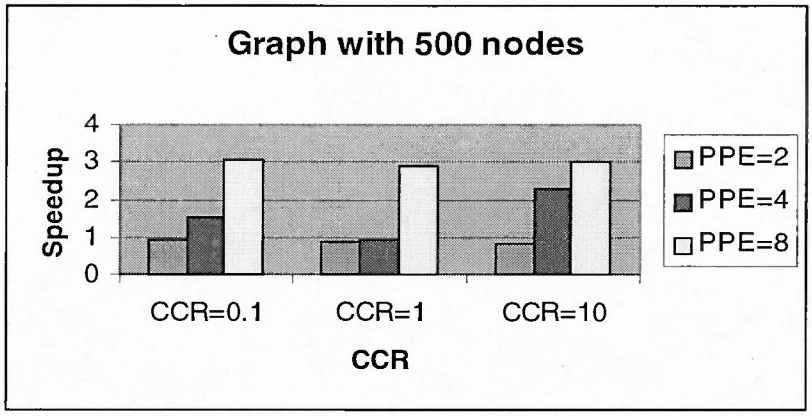


Figure 5.2 Speedup of Vertical algorithm with 500 nodes graph

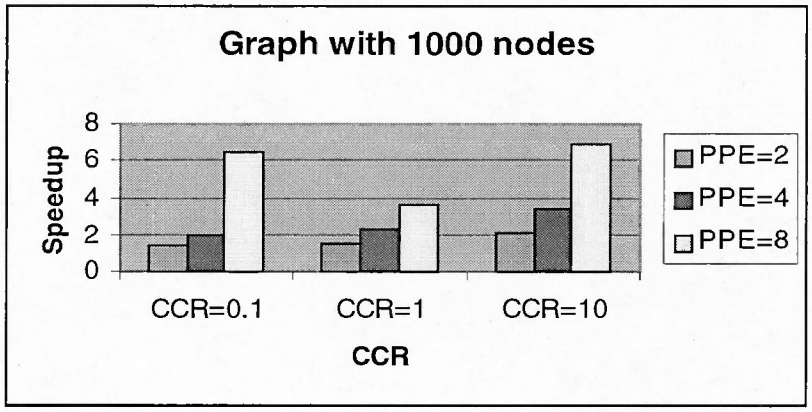


Figure 5.3 Speedup of Vertical algorithm with 1000 nodes graph

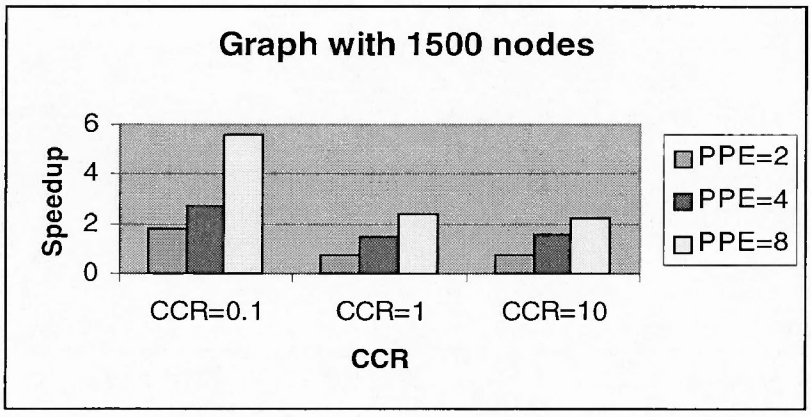
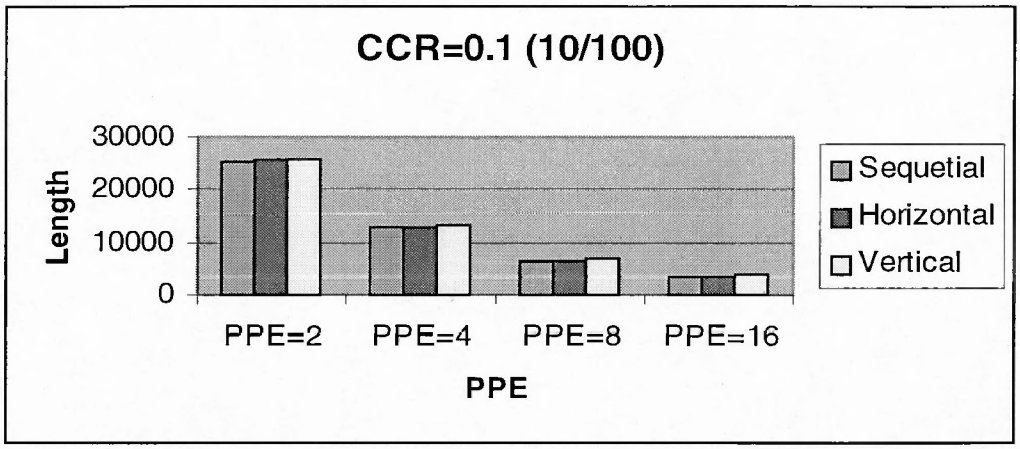
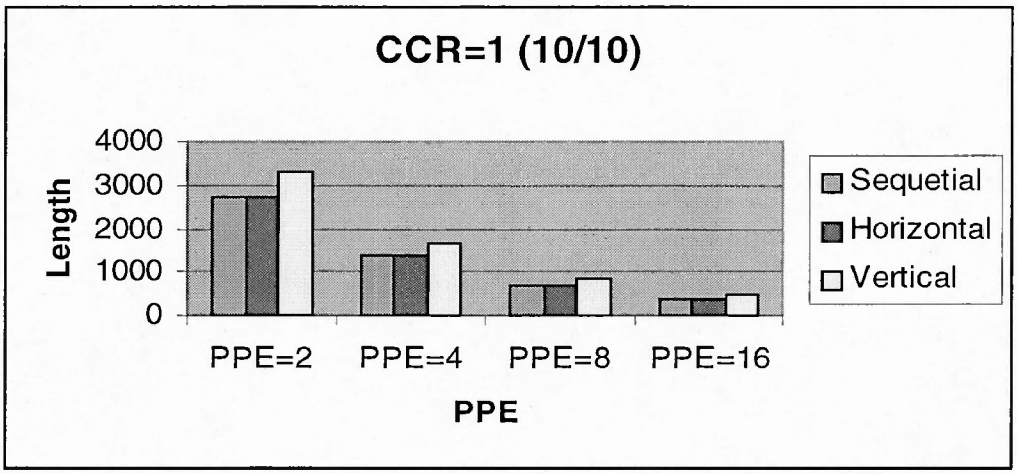


Figure 5.4 Speedup of Vertical algorithm with 500 nodes graph





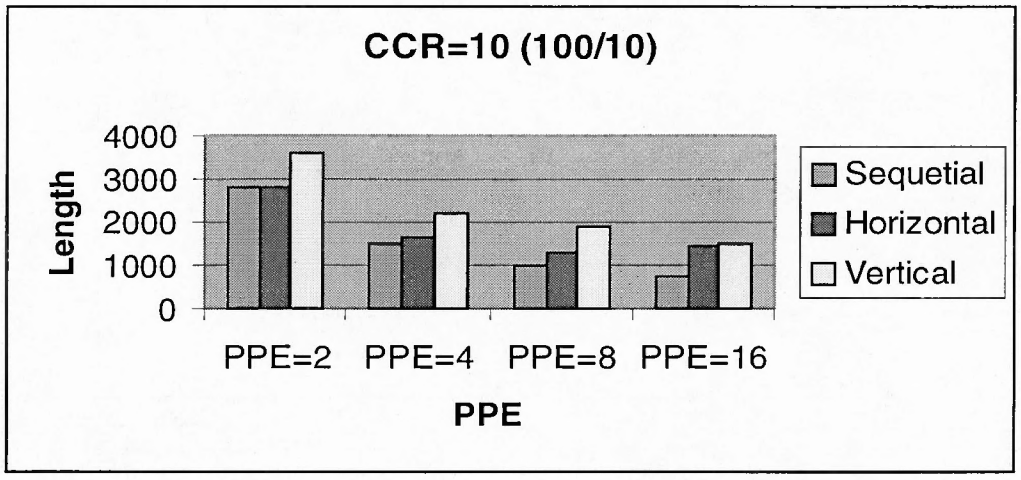
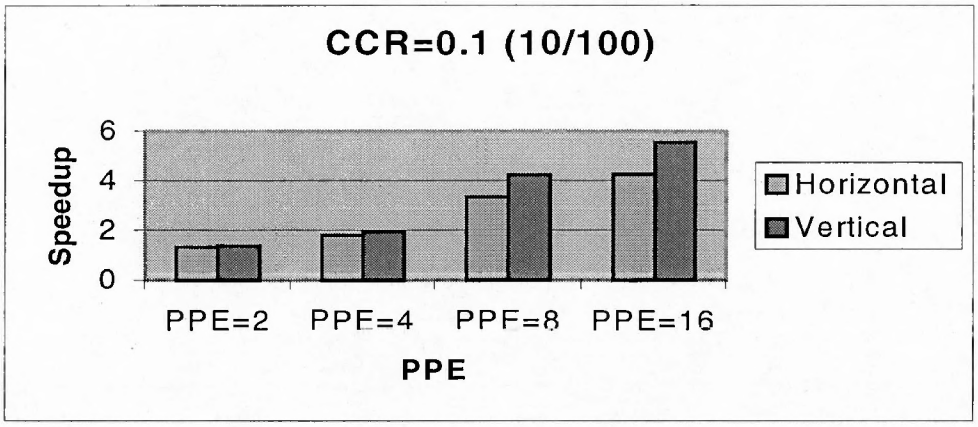
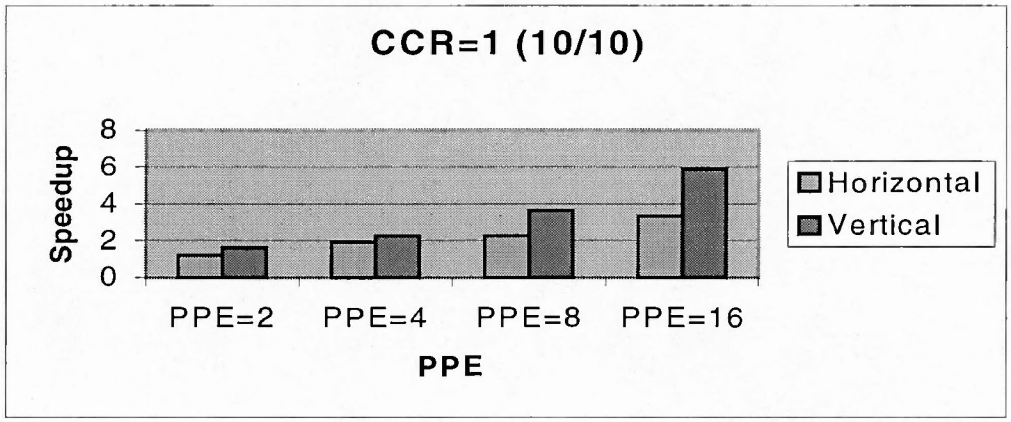


Figure 5.5 Comparison of schedule length of horizontal scheme and vertical scheme





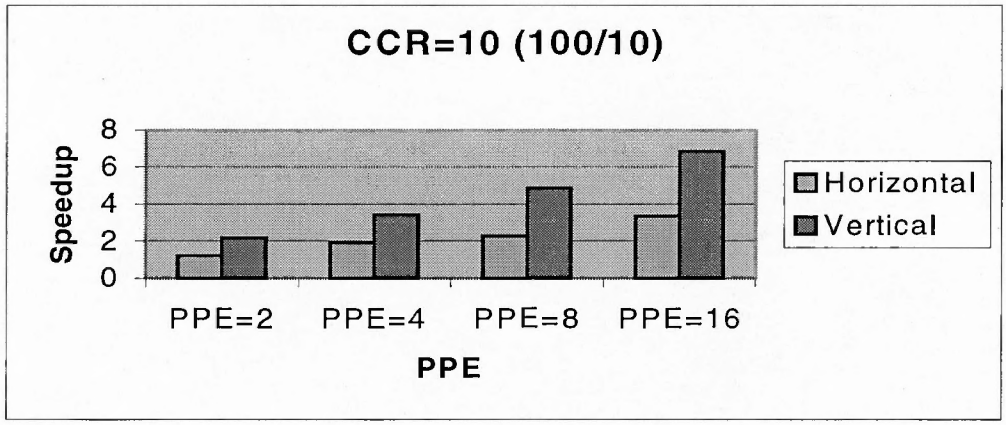


Figure 5.6 Comparison of speedup of horizontal scheme and vertical scheme

Tables 5.4, 5.5, and 5.6 show that the vertical algorithm has a better speedup than the horizontal algorithms, as seen in Figure 5.2, 5.3, and 5.4, but its schedule quality is not as good in most cases. The speedup of the vertical algorithm is better when the CCR is small.

5.4 Comparison among Different Schemes

First, we compare the horizontal scheme and the vertical scheme. The horizontal algorithm used to compare with the vertical algorithm is the one with a breadth-first partition and a simple combination since it produces the best results. Figures 5.5 and 5.6 illustrate the results of scheduling graphs of 1000 nodes and 5000 edges. The number of TPEs is the same as the number of PPEs. Figure 5.5 illustrates the comparison of schedule length between the horizontal scheme and the vertical scheme, while Figure 5.6 focuses on the speedups. From Figure 5.5, we can see that when CCR is small, both horizontal and vertical algorithms produce schedules with good schedule length. The horizontal algorithm sometimes even produces schedules that are shorter than the sequential algorithm. The reason for this is that the sequential algorithm we used here is a heuristic algorithm, not optimal algorithm. Figure 5.6 shows that when the CCR is smaller, the speedups of horizontal and vertical algorithms are closer to each other. The speedups are higher when the CCR is big, but we know that when the CCR is big, the schedule quality is lower.

We now proceed to compare algorithms among all three different schemes. In the horizontal scheme, we use an algorithm with a breadth-first partition and a simple combination since is has the best performance. In a hybrid scheme, we change the PPL

	NODE# = 1000 , EDGE# = 6000 , TPE = 64 , PPE = 16							
	SEQUENTIAL		HORIZONTAL SCHEME BREADTH-FIRST & SIMPLE COMBINATION			VERTICAL SHEME		
CCR*	TIME	LEN	TIME	LEN	S	TIME	LEN	S
0.05 (10/200)	3.57	2854	1.148	3996	3.11	0.534	4086	6.69
0.1 (10/100)	3.40	1922	0.991	2404	3.43	0.578	2518	5.88
1 (10/10)	3.45	258	0.998	318	3.46	0.503	386	6.86
10 (100/10)	3.41	985	0.848	1251	4.02	0.507	1729	6.73
20 (200/10)	3.46	1626	0.878	2232	3.94	0.494	2936	7.00

Table 5.7 Comparison of different schemes

	NODE# = 1000 , EDGE# = 6000 , TPE = 64 , PPE = 16								
	HYBRID SCHEME			HYBRID SCHEME			HYBRID SCHEME		
	8 LAYERS & PPE /LAYER = 2			4 LAYERS & PPE / LAYER = 4			2 LAYERS & PPE / LAYER = 8		
CCR*	TIME	LEN	S	TIME	LEN	S	TIME	LEN	S
0.05 (10/200)	0.891	4185	4.01	0.917	4076	3.89	0.962	3969	3.71
0.1 (10/100)	0.948	2360	3.59	0.994	2264	3.42	0.992	2260	3.43
1 (10/10)	0.948	328	3.64	0.913	329	3.78	1.015	331	3.40
10 (100/10)	0.991	1364	3.44	0.935	1457	3.65	0.981	1315	3.48
20 (200/10)	0.842	2489	4.11	0.988	2426	3.50	1.114	2459	3.11

Table 5.8 Comparison of different schemes

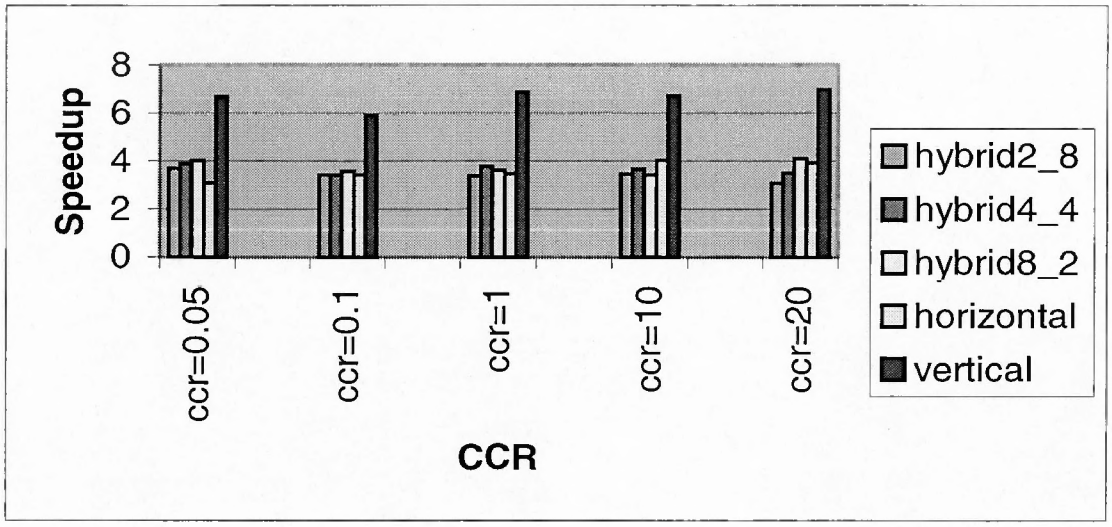


Figure 5.7 Comparison of speedups of three schemes From Table 5.7 and 5.8

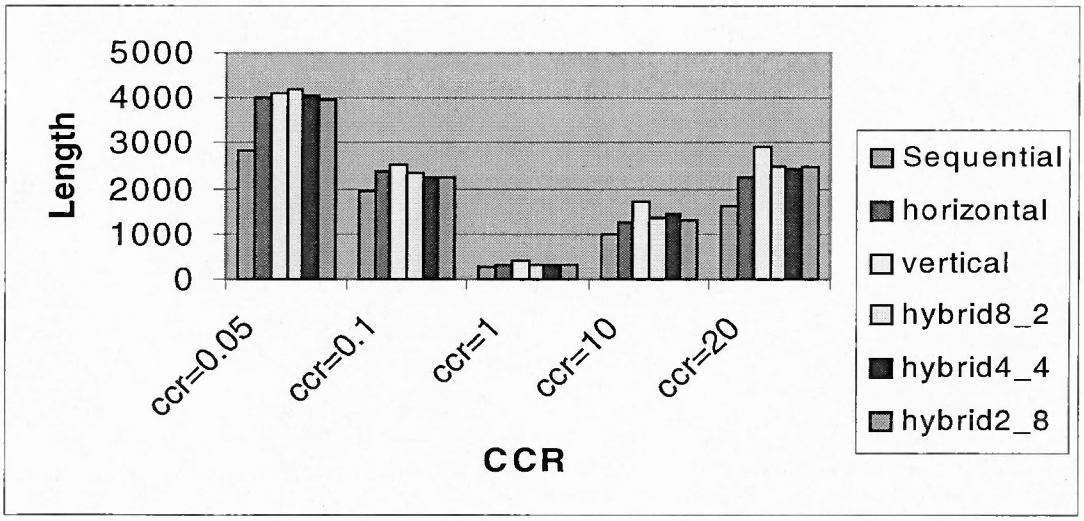


Figure 5.8 Comparison of schedule length of the three schemes from Table 5.7 and 5.8

(PPEs per layer), this results in three hybrid scheme versions, as seen in Table 5.8. In Tables 5.7 and 5.8, the task graphs are the same and have 1000 nodes and 6000 edges. The number of TPE is 64 and the number of PPE is 16. A wide range of CCR is used to compare the algorithms, from 0.05 to 20.

From Tables 5.7 and 5.8, we can also find that the vertical scheme produces good speedup values, but the output schedules are relatively longer. The horizontal scheme produces shorter schedules in most cases. The results of the hybrid scheme show that it produces better speedups than the horizontal scheme and shorter schedules as compared to the vertical scheme. We can also find cases when the hybrid scheme has a better speedup than the horizontal scheme, as well as a shorter schedule length. For example, in Tables 5.7 and 5.8, when CCR=0.05, the hybrid scheme, which has two layers and eight PPEs per layer, has better speedup, as well as a shorter schedule than the sequential scheme.

In the hybrid scheme, the choice of the parameter PPL (PPEs per layer) is critical to the performance of the algorithm. In Table 5.8, we can see that the smaller the layer, the shorter the schedule length. On the other hand, the speedup is lower when the number of layers becomes smaller.

Chapter 6

Conclusion and Future Work

The main goal of this thesis is to study parallel scheduling algorithms. Three parallel scheduling schemes were exploited, a horizontal scheme, a vertical scheme, and a hybrid scheme. The horizontal scheme based algorithms first divide the problem graph into several partitions, then these partitions are scheduled independently by a sequential algorithm, and, finally, the sub-schedules are concatenated to form the final schedule. The vertical scheme based algorithms schedule more than one node each time and conflicts are solved by allowing the node with the lowest priority to be scheduled to its sub-optimal place. Finally, the hybrid scheme algorithms combine the horizontal and vertical schemes. The hybrid scheme has a similar architecture to the horizontal scheme. However, the difference is that in the horizontal scheme we use a sequential scheduling algorithm to schedule the sub-graphs, but in the hybrid scheme, a vertical algorithm is used instead. Algorithms based on all the three schemes are simulated and compared to a sequential scheduling algorithm. The schedule quality and speedup are used to compare the performance of different parallel algorithms, as well as that of the sequential algorithm.

Based on the results of the experiments, it appears that the horizontal scheme based algorithms produce schedules with good schedule quality and the vertical scheme based algorithms have better speedup values. Nine horizontal scheme based algorithms are studied. A comparison of the performances among them shows that although horizontal

based algorithms can produce good quality schedules, there is a lot of overhead in the algorithms. Thus, the speedups of the horizontal scheme based algorithms are relatively low. The vertical scheme based algorithms, on the other hand, have better speedup than the horizontal based ones. However, the schedules produced by vertical schemes have longer schedule lengths. Therefore, the schedule quality is as good as the horizontal ones. Taking advantages of both the horizontal and vertical schemes, the hybrid scheme based algorithms can have a better speedup than the horizontal scheme based algorithms, while still getting an acceptable schedule quality.

Parallel scheduling is faster than sequential scheduling and it is able to schedule large macro data flow graphs. Parallel scheduling is still a new approach and many open problems remain to be solved before it becomes more useful in real applications. New parallel scheduling schemes, with high quality and low complexity, need to be developed. New parallel scheduling algorithms will be presented by exploiting the parallelism of existing sequential scheduling algorithms to the existing three parallel scheduling schemes. Introducing new partition and combination methods can reduce the overhead of the horizontal algorithms and increase the speedup. All the parallel scheduling algorithms studied eventually will use the sequential scheduling algorithm inside the algorithms. In the future, pure parallel scheduling algorithms will be developed.

This thesis studied parallel scheduling by simulation. For future research, it would be worthwhile to implement the parallel scheduling algorithms over some real multi-processor machines.

REFERENCE

1. T. L. Adam, K.M. Chandy, and J.R. Dickson. A comparison of list scheduling for parallel processing system. *Communications of ACM*, 17(12)pp685-690, December 1974.

2. I. Ahmad and Y.K. Kwok. A parallel approach to multiprocessor scheduling. In Int'l Parallel Processing Symposium, pp289-293, April 1995.

3. I. Ahmad, Y.K. Kwok and M.Y. Wu. Performance comparison of algorithms for static scheduling of DAGs to multiprocessors, In Second Australian Conference on Parallel and Real-time system, September 1995

4. J. Bazter and J. H. Patel. The LAST algorithm: A heuristics-based static task allocation algorithm. In *Int'l Conf. on Parallel Processing*, volume II, pp 217-222, August 1989.

5. Y.C. Chung and S. Ranka. Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors. In *Supercomputer '92*, November 1992.

6. E.G. Coffman, *Computer and Job-Shop scheduling Theory*, Wiley, New York, 1976

7. H. El-Rewini, T. G. Lewis and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, 1994

8. H. El-Rewini and H. Ali, On the scheduling problem with communication, Technical Report, University of Nebraska at Omaha, 1993

9. H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, June 1990.

10. H. El-Rewini, Task partitioning and scheduling on arbitrary parallel processing systems. Ph.D. Thesis, Department of Computer Science, Oregon State University, 1990

11. H. Gabow. *An Almost Linear Algorithm for Two-processor Scheduling.* J. ACM, 29, No. 3, 1982, pp766-780

12. M. Garey, D. Johnson, R. Tarjan, and M. Yannakakis, Scheduling Opposing Forests, SIAM J. Alg. Disc. Math., 1, no.1 , pp 72 79, 1983.

13. M.R. Gary and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

14. A. Gerasoulis and T. Yang, A Comparison of Clustering Heuristics for Scheduling DAG's on multiprocessors, *Journal of Parallel and Distributed Computing*, volume 26, no. 6, pp 276-291, December 1992.

15. D.S. Hochbaum and D.B. Shmoys, Using Dual Approximation Algorithms for Scheduling Problems: Theoretical and Practical Results. Journal of the ACM, 34(1), pp 144-162, January 1987.

16. T. C. Hu. *Parallel Sequencing and Assembly Line Problems.* Operations Research 9, No. 6, 1961, pp841-848

17. J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing.* 18(2) pp244-357, April 1989.

18. H. Kasahara and S. Narita, Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. IEEE Trans. On Computers, volume C-33, pp 1023-1029,, November 1984.

19. A.A. Khan, C.L. McCreary, and M.S. Jones. A comparison of multiprocessor scheduling heuristics. In *Int'l Conf. on Parallel Processing*, volume II, pp 243-350, August 1994.

20. W.H. Kohler, A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems. *IEEE Trans. On Computers*, volume C-24, pp 1235-1238, December 1975.

21. B. Kruatrachue. Static Task Scheduling and Grain Packing in Parallel Processor Systems. Ph.D. thesis. Dept. of Electrical and Computer Engineering, Oregon State University, Corvalis, 1987.

22. C. Papadimitriou and M. Yannakakis. *Scheduling Interval-ordered Tasks.* SIAM Journal of Computing, 8, 1979 , pp405-409

23. M. Prastein, Precedence-constrained scheduling with minimum time and communication, M.S. Thesis, University of Illinois at Urbana-Champaign,1987

24. V. Sarkar. Partitioning and Scheduling Parallel Programs for Multiprocessors. The MIT Press. 1989.

25. B. Shirazi, M. Wang and G. Pathak, Analysis and Evaluation of Heuristic Methods for Static Scheduling, *Journal of Parallel and Distributed Computing*, no. 10, pp 222-232, 1990.

26. G.C. Sih and E.A. Lee, A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processing Architectures. *IEEE Trans. On Parallel and Distributed Systems*, volume 4, no. 2, pp 75-78, February 1993.

27. J. Ullman, NP-complete scheduling problems, *Journal of Computer and System Sciences*, pp 384-393, October 1975.

28. Min-You Wu, On Parallelization of Static Scheduling Algorithms

29. T. Yang and A. Gerasoulis, A Fast Static Scheduling Algorithm for DAGs on a unbounded Number of Processors. Proceedings of *Supercomputing'91*, pp 633-642, November 1991.