SCHEDULING TASKS WITH CONDITIONAL AND PREEMPTIVE ATTRIBUTES ON A PARALLEL AND DISTRIBUTED SYSTEM

Lin Huang, M.Sc.

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN THE DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ADELAIDE

January 1999

Abstract

Parallel programming aims to make maximal use of computing resources in parallel and distributed systems. It is concerned with a number of issues which are not encountered in its sequential programming counterpart. To list a few, these include task partitioning, task communication and synchronization, task scheduling and system performance. Such issues must be efficiently handled in order to achieve high system performance.

This thesis discusses the scheduling problem and concentrates on its various forms in parallel processing. It is concerned with the distribution of the tasks of a parallel program onto the underlying available processors. The objective is the optimization of parallel program execution time.

This thesis studies a variation of the task scheduling problem, termed *conditional task scheduling*, which is characterized by conditional task spawn and intertask communication events within the parallel program. The corresponding program is denoted a *conditional parallel program.* In conditional task scheduling, the production of a task model which represents the corresponding parallel program is practically impossible to precisely obtain prior to program execution, and may vary between different executions. A scheduling policy which achieves high performance in one execution does not necessarily result in the same efficiency for other executions.

The strategy by which this thesis deals with conditional task scheduling takes such factors into consideration. A revised conditional task model is introduced to reflect the conditional spawn and conditional communication between tasks. Approaches to the construction of the task model prior to program execution is also presented. A new scheduling algorithm, CET, is proposed which takes as input a statically-constructed task model and produces a scheduling policy by which the tasks of the conditional parallel program can be efficiently distributed onto the underlying processors. Simulation is utilized to show the performance and properties of the strategy.

This thesis examines another variation of the task scheduling problem, namely, preemptive task execution and preemptive task scheduling. Preemption may occur when the communication between parallel tasks is permitted to occur at any point within the tasks, rather than restricted to the beginning or the end of the task. Strategies are proposed in this thesis to guarantee performance improvement in preemptive task execution over non-preemptive task execution. With regard to preemptive task scheduling, a preemptive task model is presented to illustrate the preemption between parallel tasks. This thesis also puts forward a preemptive task scheduling algorithm, PET, to deal with the distribution of parallel preemptive tasks. The construction of the preemptive task model prior to program execution adopts a similar method to that of the conditional task model. Experiments indicate significant performance improvement when considering preemption in task scheduling.

This thesis also investigates support for parallel programming, in particular, those aspects involved with conditional task spawn and conditional intertask communication. This is termed *conditional parallel programming.* A runtime library is presented and made available to programmers to simplify the development of parallel programs.

This thesis introduces an environment, called *ATME*, which realizes the automation of the (conditional and preemptive) task scheduling process. *ATME* provides explicit support for conditional parallel programming. Using *ATME*'s self-contained runtime library, the programmer is relieved of the need to consider operational issues which have little to do with the application problem itself. Taking as input the user-partitioned tasks of a parallel program which employs the *ATME* runtime primitives, the *ATME* environment instruments the user tasks to collect runtime task information. *ATME* also automatically generates the task interconnection structure. *ATME* then constructs the task model prior to execution, and statically generates a scheduling policy. After the program completes its execution, *ATME* collects the runtime-generated trace files which contain task information and uses them to predict the task model for the forthcoming program execution. In addition, *ATME* aims to provide tuning suggestions for both the application programmers and *ATME* itself, so as to improve the parallel program design.

Declaration

This is to certify that this thesis contains no material which has previously been accepted for the award of any degree or diploma in any University. To the best of my knowledge and belief it contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

If this thesis is accepted for the award of the degree, permission is granted for it to be made available for loan and photocopying.

Lin Huang

January 1999

Acknowledgments

I wish I could thank all the people who kindly offer me help and concern during my study as a Ph.D candidature. First of all, I am deeply indebted to my supervisor, Dr. Michael J. Oudshoorn, for his invaluable advice, guidance and full support. I also wish to thank to Dr. Jiannong Cao, who acted as a joint supervisor before leaving The University of Adelaide, for his bright instruction and kind encouragement in the early stages of my study. I would like to thank University of Adelaide, for granting me a University of Adelaide Scholarship (UAS) and an Overseas Postgraduate Research Scholarship (OPRS), so that I could complete my study without any financial difficulty. I would like to show my heartful gratitude to Department of Computer Science, headed by Prof. Chris Barter, for providing me with a good research environment and strong financial support to attend conferences, in particular, those overseas such as in The United States.

Others whose comments, technical and academic assistance contributed to shape this work to varying degrees include Dr. Hesham El-Rewini, Dr. Weiping Zhu, Bob Fang and Stuart Beck.

Many thanks to all my friends who are always ready to extend their hearts to share with me the difficulties and the happiness. Special thanks are presented to Dr. Puquan Ding and Mrs. Gu Xu who initially encouraged me to apply for the highly competitive

OPRS scholarships, and offered me parenthood love and concern. Thanks also are given to my fellow postgraduates, Zhengyi Wu, Hendra Widjaja and Kevin Lew, with whom I share the joys and pains of study and work.

I wish I could express all my gratefulness to my dear parents, Mr. Kun Huang and Mrs. Bin Zou, who have dedicated themselves to a career in education in China for more than 35 years. I thank them for their bringing me up, and for their understanding, instruction and support through all these years. I wish I could say that I have fulfilled the dream which they held when they were my current age, but which they were unable to realize due to various reasons beyond their own control then.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Advances in computer architecture have been dramatic in recent decades: from the original uniprocessor system to various types of parallel systems. Such a change is promoted by both the technology of computer hardware and the requirements of practical applications. On one hand, new parallel and multiple processor architectures are continuously designed, such as the iPSC from Intel, Sequent Balance & Symmetry, Ncube/ten, and CM-5 from Thinking Machines [65, 180]. Furthermore, with the swift development of low-cost computers and high-bandwidth, low-latency communication networks, it has become commercially viable to connect a number of computers together to form a parallel and distributed system. New architectures and systems, with the aim of increasing both system (machine) and application program performance, have broken the monopoly held by sequential computer architectures since they were initially put forward by Von Neumann in 1946. These parallel systems make it possible to efficiently pursue solutions to complex and large-scale problems, which may take several magnitudes of time longer if computed on a sequential machine.

The other factor propelling the advancement of computer architectures is that more and more applications emerge with a demand for efficient processing. Such

applications can be found in areas including scientific computation, image processing, modeling, analysis and simulation [80]. Applications in other areas, such as office automation and resource management, also demand highly efficient processing of their requirements [65]. The application programs are becoming increasingly complex and require a number of components to cooperate with each other to solve a specific problem. Parallel systems and associated software appear to suit this demand. In these circumstances, a program is partitioned into a number of tasks which are then scheduled onto the underlying available processors and executed in parallel. During execution, tasks communicate and synchronize with one another to pass necessary data between themselves.

Motivated by both widely available parallel systems, and demands for efficient processing, parallel programming has become a straightforward solution to complex (especially large-scale) problems, with the objective of maximizing system and program performance. Scientific computation and image processing are two examples which significantly benefit from parallel programming.

By introducing parallelism into programming, it is possible to make great use of computing resources. However, parallel programming also raises new issues which must be efficiently solved before it is put into widespread practice. This thesis is concerned with two of these issues:

- the performance enhancement of parallel systems; and
- the support for programmers in parallel program development.

There is no doubt that parallel processing brings about performance improvement of application programs and parallel systems; however, at the same time, it also incurs additional overheads. The overhead may come from a variety of sources, including

the improper design of parallel algorithms, execution delay due to communication and synchronization between tasks, and competition for shared resources. As a result, linear speedup in terms of system performance can not be practically achieved in a parallel system. That is to say, an increase in the number of processors in the system is not directly proportional to the magnitude of system performance improvement. As the number of processors comprising the parallel system increases, overheads associated with parallel processing increase correspondingly. This offsets the benefit brought about by the parallel system and parallel algorithm. This is illustrated as the "saturation effect" by Chu [42]. Consequently, effort is required to make in order to extract all potential benefits of the parallel system, so as to enhance overall system performance.

It is commonly regarded that parallel programming is intrinsically difficult, in comparison to its sequential counterpart, owing to the complexity of the partitioning of a program into tasks, task scheduling, task communication and synchronization [52]. Writing a high-performance parallel program demands expert knowledge of both the system and relevant programming techniques. It is claimed that parallel programming environments or tools can significantly reduce a programmer's workload and increase system performance by automating tedious chores [179]. At present, a number of programming support tools and environments have been presented to relieve programmers of the complexities of parallel programming. Parallel algorithms have also been examined to help solve these practical problems on parallel systems.

The task scheduling problem in parallel processing is related to both issues mentioned above, i.e., parallel system performance and parallel programming support. On one hand, the appropriate scheduling of parallel tasks onto underlying processors can result in significant improvement in system performance. It can also reduce the

runtime overhead by decreasing the communication overhead between tasks on different processors. An efficient task scheduling policy can distribute two communication-intensive tasks onto the same processor, thereby reducing extra runtime overhead by eliminating the need to wait for data arrival, and consequently upgrading system performance. On the other hand, task scheduling is a critical step in parallel programming. Each task must be allocated onto a particular processor before it can be physically executed. The process of task scheduling can be automated so as to largely free application programmers from the complexities associated with parallel programming [52].

This thesis studies two types of task scheduling problems: the conditional scheduling problem and the preemptive scheduling problem. Briefly, conditional task scheduling examines the scheduling of parallel tasks in which runtime operations (primarily task spawn, data transmission and data reception) can be associated with conditional branches. This case is commonly encountered in application programs, but has drawn little attention to date [51]. Preemptive task scheduling, on the other hand, concentrates on the scheduling of tasks in which communication operations are not necessarily restricted to the start or the end of the task. This problem has been commonly neglected by most current scheduling research, which mainly focuses on non-preemptive task scheduling, as reviewed in Chapter 2. Preemption in program execution is also investigated in this thesis. Both research topics aim to improve the performance of the parallel system through an adequate distribution of parallel tasks, i.e., task scheduling.

This thesis examines support for parallel program development. An environment, named $ATME$, is developed to automate the scheduling of conditional and preemptive parallel tasks. The environment also realizes appropriate support for parallel program

development, through a runtime library.

Section 1.1 of this chapter explains the context of the work, within the broader picture of parallel and distributed computing. The problems addressed by this thesis are stated in Section 1.2; this is followed by Section 1.3 which discusses the thesis contribution. Chapter 1 is concluded in Section 1.4 which describes the thesis organization.

1.1 Context

This section describes the context of the thesis. Detailed background and related work is presented in Chapter 2. Section 1.1.1 illustrates parallel systems, as well as application programs, on which this thesis is developed. Section 1.1.2 discusses the process involved in the general task scheduling problem.

1.1.1 Systems and Applications

This thesis bases its study on a *multiple processor architecture* in which processors are loosely-coupled, run in parallel and communicate with each other via explicit message-passing through networks. Such systems are also categorized as *loosely-coupled parallel and distributed systems* [109]. With the development of high-speed, low-latency communication networks, and the low cost of computer hardware, such a multiprocessor architecture has become practically viable to solve application problems cooperatively and efficiently.

This thesis deals with the *task-level parallel processing* of application programs. A parallel program is regarded as composed of a number of interrelated tasks. Each task is an independent (sub)program which realizes relatively complicated functionality (part

of the application requirements). The execution load of a parallel task is not necessarily the same across different program executions. Similarly, the communication pattern and volume of data communicated between tasks is not necessarily the same across different executions. Each component task in the parallel application is regarded as an atomic action and an atomic scheduling unit. Such parallel applications can be either fined-grained or coarse-grained, depending on the granularity value which is defined as the average ratio between task computation and inter-task communication magnitude of the parallel program [180]. The ratio is denoted as $AvePMRatio$ in this thesis. Correspondingly, parallel applications can be categorized into three groups: computation-intensive ($AvePMRatio \gg 1$), neutral ($AvePMRatio \approx 1$) and communication-intensive ($AvePMRatio \ll 1$).

This thesis focuses on the study of the *static scheduling* (i.e., the scheduling policy by which to distribute parallel tasks onto underlying available processors is produced prior to program execution) of parallel tasks onto the loosely-coupled parallel and distributed system. The problem is formally stated in Section 1.2. The scheduling problem has been proved to be NP-complete [168], i.e., no optimal solutions are available within polynominal computation complexity. Heuristics have been proposed to address different variations of the task scheduling. For instance, with and without attention to task precedence relationships, and restrictions on the program and systems, such as 2-processors, 1-unit of execution time etc [51]. A detailed review of related work is found in Chapter 2.

This thesis assumes that there is only one parallel and distributed program exclusively utilizing the resources of the underlying parallel system. That is to say, the work discussed in this thesis excludes the consideration of the multi-user task scheduling.

1.1.2 Task Scheduling Process

This section explains the general process involved in task scheduling. The scheduling problem can be described as a "resource-consumer" problem [52], where it assumes a set of resources (processors) and a set of consumers (parallel tasks of an application program) served by available resources according to a certain scheduling policy. The objective of the scheduling problem is to search for an efficient scheduling policy by which consumers can utilize resources at hand to optimize the desired performance measure, such as the minimal processing time.

The task scheduling problem can be decomposed into three major functional components: the *task model* (which portrays the constituent tasks and intertask relationships within a parallel program), the *processor model* (which describes the configuration of the parallel and distributed system), and the *scheduling algorithm* which produces a scheduling policy by which tasks of a parallel program are distributed onto available processors and arranged into execution commencement order for tasks assigned to the same processor. All of these three components are determined by the *scheduling objective* which is the performance measure to be optimized.

In this thesis, the *task model* is depicted by a *DAG* (Directed Acyclic Graph) in which graph nodes represent parallel tasks, and directed arrows represent inter-task precedence relationships. Two tasks with such a precedence relationship is named a *parent* and a *child task*, respectively. In this thesis, a parent task has a static relationship to its child task. The term *parent task* does not necessarily mean that it actually spawns the child task at runtime. It is possible for a task to have more than one parent tasks as shown in Figure 1.1. Here it is seen that task E has as parent tasks, the tasks A and C. Tasks and task interconnections in such a graph are

associated with attributes representing task behaviour such as computation time and communication time. The formal definition of a *DAG* is given in Section 3.2. Other task model representations are discussed in detail in Chapter 2.

The *processor model* abstracts over the architecture of the underlying parallel and distributed system on which the parallel program is to be executed. Such a processor model is usually illustrated by a undirected graph, with associated attributes representing the processing speed of processors as well as the data transfer rate along communication networks. Section 3.1 gives the formal definition of the processor model.

The *scheduling algorithm* aims for an efficient scheduling policy to optimize the desired performance measure (scheduling objective). Here, *efficiency* is discussed in terms of the *effect* of the scheduling policy on system performance, rather than the *process* of achieving the policy. This thesis adopts *parallel execution time* of the program (abbreviated as PT) as its scheduling objective.

An example of the entire process of task scheduling is illustrated in Figure 1.1. The scheduling algorithm takes as input a task model and a processor model, and generates a scheduling policy to distribute user tasks. Suppose the parallel and distributed system is composed of identical processors which are fully connected by identical networks. In the task model, the superscript to the task name represents the execution time of that task on a processor. Values along the edge between two tasks indicates the communication time if these two tasks are allocated on separate processors; it is assumed that the communication time is 0 if the two tasks are assigned onto the same processor. The scheduling policy is represented by a chart — with the *processor* axis showing all processors available in the parallel system, and the *time* axis illustrating the execution order of tasks assigned to each processor. Tasks execute concurrently, but communicate with each other at specific points to transfer data. Communication

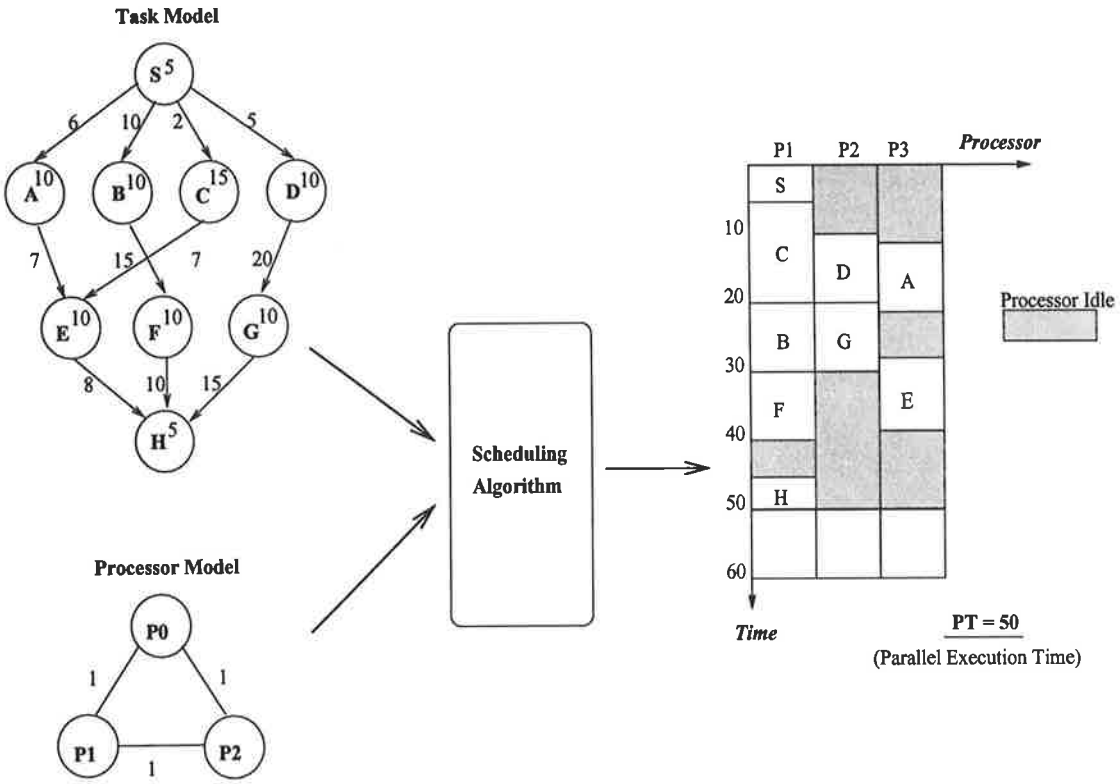


Figure 1.1. The procedure of solving a task scheduling problem.

between tasks on two separate processors incurs runtime overhead which may delay the execution of the task. The system and program performance is measured by *parallel execution time* which is the wall-clock completion time of the last task of the parallel program.

1.2 Problem Statement

The thesis studies the task scheduling problem in parallel and distributed systems, particularly focusing on two practical cases: conditional task scheduling, and preemptive task execution and scheduling, which have not been investigated extensively to date. These problems are stated in Section 1.2.1 and Section 1.2.2, respectively. The thesis also investigates robust support for parallel program development, as introduced

```
void A ()
{
    ...

  if  (condition "a" is satisfied)  {
    if  (task E has not yet spawned)
      spawn task E;
    send data to task E;
  }
    ...
}
```

Figure 1.2. Segment of pseudo code of task A in Figure 1.1.

```
void E ()
{
    ...

  if  (task A exists and has transmitted data to me)
    wait for the arrival data from task A;
  if  (task C exists and has transmitted data to me)
    wait for the arrival data from task C;
    ...
}
```

Figure 1.3. Segment of pseudo code of task E in Figure 1.1.

in Section 1.2.3.

1.2.1 Conditional Task Scheduling Problem

With respect to conditional task scheduling, application programs with which this thesis deals have the following common characteristics: runtime operations (in particular, task spawn, data transmission and data reception) in each task of a parallel program may be associated with conditional branches. The value of the conditions on a branch is not determined until runtime, depending on, say, initial parameters and user interaction. Such a parallel program is termed a *conditional parallel program*

in the thesis. Figures 1.2 and 1.3 present fragments of pseudo code for a pair of communicating parent and child tasks (task A and E), as displayed in Figure 1.1. Here, it is supposed that conditions are associated with the interaction between tasks A and E, as well as tasks C and E, in Figure 1.1. It can be observed that applications in which no conditional branches are associated with task runtime operations (i.e., deterministic task execution, in which task behaviours can be precisely determined prior to execution) are a special case of the applications addressed in this thesis.

The study on the conditional parallel program and conditional task scheduling in this thesis is stimulated by actual user requirements. If the application software is regarded as a "black box", then for each program execution, the program receives a set of inputs and returns a set of outputs. Each set of inputs to the program represents a "usage pattern" of the software. During the lifetime of the application, different users, or even an individual user, may have different usage patterns, that is to say, different ways of interacting with the application program. Consequently, the tasks involved in interactions at runtime may not be identical across all executions. In addition, all task interconnections may not be fully exercised in every execution either. This situation is reflected in the program source code as the conditions (including loop conditions) associated with task runtime operations.

Conditional programming is not an outstanding issue in many areas of scientific computing because their communication patterns are mostly fixed and the processing of each task is normally statically well specified. However, it is quite a common and interesting problem in other areas where parallel tasks vary from each other (i.e., not all tasks are identical) and are relatively large in terms of computation magnitude. Each task may be conditionally invoked by a number of other tasks, and may also conditionally communicate with other tasks.

Conditional task scheduling, following the pattern of the general task scheduling, needs to address four aspects: the scheduling objective, the task model, the processor model and the scheduling algorithm, as stated in Section 1.1.2. In addition, it is assumed that task runtime operations only take place at the beginning and end of the task, i.e., a task does not commence its execution until it receives desired data; it runs until completion and then transmits all data to its succeeding child tasks. That is to say, in the study of conditional task scheduling, the parallel program is assumed to be non-preemptive. Conditional task scheduling is concerned with more complicated issues than its deterministic scheduling counterpart. These include two aspects: prior to program execution, the construction of a task model which approximates the actual model in the forthcoming program execution; and a scheduling algorithm which deals with the conditional task model. Due to variations in the task model between different executions, the scheduling policy adopted should vary accordingly, with the aim of achieving good system performance in most executions. Extensive research has not been undertaken on the conditional task scheduling problem. At present, there is a lack of scheduling algorithms or tools to tackle this form of task scheduling.

1.2.2 Preemptive Task Execution and Scheduling Problem

The second problem studied in this thesis is preemptive task execution and preemptive task scheduling. The preemption problem in parallel processing occurs when message-passing operations, as well as task spawn, are permitted to take place at any point in the task. Such operations are no longer restricted to the start or the end of the task as is commonly assumed in most current non-preemptive scheduling research. Consequently, the child task may commence execution prior to the completion of its parent task(s).

The thesis divides the preemption problem into two sub-problems: preemption task execution (denoted as *PTE*) and preemptive task scheduling (denoted as *PTS*). The *preemptive task execution* problem is concerned with performance achievement merely through the occurrence of preemption at runtime. It supposes all tasks have already been scheduled onto processors via a particular scheduling algorithm (irrespective of whether it considers preemption or not during the distribution of tasks). Intuitively, PTE may either improve or degrade system performance; the thesis intends to propose an approach to guarantee a performance gain.

On the other hand, the *preemptive task scheduling* problem is basically involved with the study of the scheduling algorithm itself. The algorithm takes preemption into consideration when distributing parallel tasks onto underlying available processors. Other issues, such as task model construction, are also addressed, with the aim of assisting the scheduling algorithm to produce an efficient scheduling policy for the preemptive parallel program. This thesis proposes a new scheduling algorithm to deal with preemption in program execution. This thesis also analyzes system performance gained through preemption.

It may be argued that the preemptive task scheduling problem can be converted into the non-preemptive task scheduling case, after which current algorithms can be applied and thus solve the problem. A straightforward solution may partition the original (preemptive) parallel task into a number of sub-tasks, in each of which communication occurs only at the beginning and end of the sub-task. As a result, all sub-tasks are non-preemptive, and therefore the scheduling algorithms for non-preemptive parallel programs can be applied. However, new problems are raised by this approach. To list a few, firstly, further task partitioning introduces more precedence relationships and communication among tasks and sub-tasks, over and above the original task model.

Therefore, it can be forseen that additional overhead is incurred at runtime. Secondly, additional constraints (such as sub-tasks of the same task are best distributed onto the same processor in order to reduce interprocessor communication) must be considered by the scheduling algorithm. Subsequently, modifications to current scheduling algorithms are still required in order to meet these new criteria. Finally, the partitioning of a task into a group of sub-tasks is not always feasible. For instance, when message-passing operations are incorporated into branches with other computation, the separation of communication from computation is not easy to practically deal with. New approaches must be investigated so as to efficiently tackle the preemptive task scheduling problem.

When discussing the preemption problem, for the sake of clarity, the thesis assumes that a "deterministic task model" is available. That is to say, when studying the preemptive problem, no consideration is given to the situation in which conditional branches are associated with task runtime operations. This thesis proposes strategies for the conditional scheduling and preemptive scheduling problems respectively. The proposed strategies can be combined together to solve the mixed problem of conditional and preemptive scheduling, as realized in the environment $ATME$ (discussed below).

1.2.3 Conditional Parallel Programming Support

On the basis of the research conducted on the conditional task scheduling problem, as well as the preemptive task execution and scheduling problem, the thesis introduces $ATME$, a parallel program development environment, to realize the automation of task scheduling processes. Moreover, $ATME$ assists the process of program design and implementation, through a set of runtime primitives, with an awareness of the new challenges encountered in conditional and preemptive parallel programming.

Consider the example of the message-passing operations in a conditional parallel program, as illustrated in Figures 1.2 and 1.3. It is clear that an extra burden has been imposed on application programmers who must deal with issues which are beyond the scope of the application problem itself. The extra work deals predominantly with *conditional task spawn* and *conditional message-passing* operations.

Task E, in Figure 1.1, is conditionally spawned by its two parent tasks A and C. When task A spawns task E, which may also be spawned by its other parent task C, task A is required to check whether task E has already been spawned or not, since task E can be spawned once, and only once. With current programming support such as PVM [69], the parent task A has no straightforward way of knowing the status of its child task E. Extra work for application programmers is therefore incurred through conditional parallel programming. As observed, these issues are mostly related to program implementation. This thesis believes that application programmers should focus on the functionality of the program, rather than tedious protocol enforcement. The programmer therefore demands support with respect to conditional spawning of tasks.

Furthermore, task E conditionally receives data from its parent tasks A and C. In the case where task A is not spawned by its parent task S (Figure 1.1), task E should not wait for data from task A since it will clearly never arrive. Existing programming support does not provide task E with facilities to detect the status of its parent tasks. Application programmers have to build their own functions to realize such behaviour.

At present, application programmers are provided with a number of environments or tools, as well as runtime libraries, which aim to ease the difficulties of the development of parallel applications, as reviewed in Chapter 2. However, so far there is no direct and powerful support available for conditional parallel programming.

Such support must be easy to use and flexible. It is necessary to cater for the various needs of programmers so that those who wish to tackle low-level issues in programming have appropriate support, as do those who wish to have the process automated as much as possible.

1.3 Contribution

This thesis proposes a practical strategy to tackle conditional parallel task scheduling problem. The thesis introduces a conditional task model to represent the application program in which task runtime operations may be attached with conditional branches. The approach does not incur significant runtime overhead, since most of the work is undertaken statically prior to program execution. The approach is composed of two primary aspects: the construction of the task model for the forthcoming execution, based on execution profiles; and a conditional scheduling algorithm to deal with the conditional task model and produce an efficient scheduling policy for the application program. The thesis develops detailed discussion and presents experimental results, regarding the conditional task scheduling problem.

This thesis studies the preemption problem in parallel programming by focusing on two aspects: the preemptive task execution and the preemptive task scheduling problem, as stated in Section 1.2.2. The thesis presents a strategy to guarantee a performance gain through preemptive task execution. The thesis also proposes a preemptive task model to illustrate preemption in the parallel program. A preemptive task scheduling algorithm is presented to generate a scheduling policy for the preemptive task model. The thesis further investigates the performance improvement brought about by the preemptive task execution and preemptive task

scheduling.

This thesis examines the support for parallel program design and implementation. The thesis presents the application programmer with an environment, named $ATME$, which automates the task scheduling process and explicitly supports conditional parallel programming. The programmer, therefore, does not have to deal with operational issues outside of the application problem itself. $ATME$ is an open environment, and all its functional components are deliberately designed with a clear user interface. Consequently, the latest research on any aspect of $ATME$ can be easily plugged in to provide more powerful support for application programmers.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 briefly states the background of this work and reviews the related work. The conditional task scheduling (CTS) problem is studied in Chapter 3. A conditional task model is formally defined, with the virtue of describing the conditional task runtime operations. Chapter 3 also presents a conditional task scheduling algorithm, named CET, to schedule conditional tasks onto the underlying parallel and distributed system. Chapter 4 studies the preemptive task execution and preemptive task scheduling. The performance achievement of preemptive execution is addressed in detail. In addition, an algorithm, named PET, is proposed to deal with the preemptive task scheduling problem. Conditional parallel programming support ($CPPS$) is studied in Chapter 5, where a runtime library is introduced and design issues are discussed. Chapter 6 presents the framework of the $ATME$, which aims to automate task scheduling and support the (conditional) parallel program development. The combination of CET and PET in

ATME provides a solution to the conditional and preemptive scheduling problem. Design and implementation details are also discussed. Chapter 7 describes the simulation undertaken in the work, and experimental results regarding conditional and preemptive task scheduling problems discussed. The thesis is concluded in Chapter 8 where the work is summarized and future directions are presented.

Chapter 2

Background And Related Work

Parallel and distributed systems have gained wide acceptance due to their virtues such as high system performance, high reliability and efficient system utilization. Application areas include simulation, engineering modeling, scientific computation, image processing, artificial intelligence and financial analysis etc., where execution efficiency and/or response time are essential.

This chapter reviews the research work with regard to parallel systems, in particular, the task scheduling problem. Section 2.1 classifies current parallel and distributed systems, under various taxonomies. Section 2.2 concentrates on the performance issue in parallel processing and states several speedup functions adopted in current research. It has been shown, both theoretically and experimentally, that achievement of linear speedup is impossible in a real parallel system; that is to say, doubling the number of processors does not necessarily result in a doubling of system performance. The discussion of the speedup issue also examines factors which incur various overheads in parallel processing and subsequently reduce the performance of the entire system.

This thesis regards task scheduling as a key step to improve the performance of the parallel and distributed system. Section 2.3 reviews scheduling algorithms which are

presented to deal with various forms of the scheduling problem. Programming support for the design and implementation phases of parallel program development aims to relieve programmers of the burden in developing highly efficient parallel programs. Section 2.4 studies general approaches for such programming support, and focuses on the scheduling tools to examine the presently available aids for task scheduling.

2.1 Parallel and Distributed Systems

Early parallel and distributed systems were realized by modifying the CPU in traditional computer architectures, or simply by placing additional CPUs into an existing system [109]. Advanced VLSI technology and the development of design automation tools in the seventies and eighties have removed fundamental architectural constraints and made it possible to work on a single problem via the cooperation of multiple (possibly thousands) processors [80].

This section reviews current parallel and distributed systems (Section 2.1.1), according to whether constituent processors (computers) are loosely or tightly coupled, and the communication mechanism employed by the system. At present, there lacks a consistent agreement in terms of taxonomy by which to categorize all existing systems. Researchers have proposed a number of taxonomies to classify parallel and distributed systems, as discussed in Section 2.1.2.

2.1.1 Tightly and Loosely Coupled Systems

A parallel and distributed system can be conjectured to be composed of a number of computers or processors interconnected for the purpose of parallel processing. Here, a computer, or computer system, refers to an integrated device which is capable of

accepting input data, applying a sequence of processes to the data, and producing execution results; while a processor generally refers to a CPU, but not excluding the existence of I/O processors, which deal with data communication between processors. Therefore, in this thesis, the two terms, processor and computer, are synonymous.

On the basis of whether computers are tightly-coupled or loosely-coupled within the distributed system and the communication mechanism employed, most existing parallel and distributed systems can be classified into two major groups: tightly-coupled systems with shared memory (as a communication mechanism) and loosely-coupled systems which utilize message-passing. In a tightly-coupled system with shared memory, processors are interconnected with at least one common memory address space which makes intimate interactions possible among processors [109]. Processors normally communicate and synchronize via shared variables. Such parallel systems include IBM 370/168, CDC 7600, Honeywell 60/66 and SEQUENT Balance & Symmetry [65, 109].

In a loosely-coupled system using a message-passing communication mechanism, processors are coupled via a communication network, either over a wide-area or local-area network. Each processor has its own dedicated memory, therefore, processors can run in parallel without interference. The most commonly used communication mode between processors takes place via message-passing. The processors may be formed into a number of topologies, such as linear, star, ring, mesh, hypercube, pyramid, butterfly and fully-connected (clique) [65, 180]. A typical example of a loosely-coupled system is one composed of processors interconnected by networks, as seen in ARPANET, DCS and DCN. The SAGE system (duplicated processing system), and the dynamic data flow machine Irvine (a network of multiprocessors) also fall into this category [109].

In addition, there are some existing machines which do not fall into either category mentioned above, and are therefore termed "hybrid" architectures. For instance, ELXSI 6400 [132] and the Virtual Port Memory research machine at New Mexico State University [98] are tightly-coupled systems but adopt a message-passing mechanism to deal with data communication and synchronization. Some loosely-coupled systems can also use a shared-variable mechanism to handle processor interactions. An example of this is the BBN Butterfly [47].

2.1.2 Other Taxonomies

Several other taxonomies of parallel and distributed systems have been presented as well. One of the early classifications originates with Flynn [61], who classifies all computer systems via two streams: instruction stream (a sequence of instructions processed by a computer) and data stream (a sequence of data supplied to an instruction stream). Four types of systems are therefore proposed: SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data) and MIMD (Multiple Instruction Multiple Data). SISD actually refers to sequential machines, as proposed by Von Neumann. MISD is not of great practical significance, due to severe application restrictions with respect to parallel processing. SIMD and MIMD are forms of parallel architectures. Illiac IV and OMEN 64 are SIMD machines, while the common message-passing parallel system can be regarded as MIMD. There is a new type of parallel systems originating from MIMD, SPMD (Single Program Multiple Data), which applies the same program (rather than instructions as in MIMD) onto different data sets.

Anderson *et.al.* [6] base their classification on three hardware archetypes: PEs (Processing Elements such as processors or computers), paths (a medium by which a

message is transferred between PEs) and switching elements (an entity which affects the destination of a message, between the message sender and receiver). The MERIT system [13] is classified as a complete interconnection DDC type, indicating "Direct transmission" (regarding processing elements), "Dedicated connection path" (regarding communication path), and "Complete interconnection" (regarding switching elements).

Johnson [97] divides parallel machines, according to their memory structure and data communication mechanism, into four groups: GMSV (Global Memory – Shared Variables), GMMP (Global Memory – Message Passing), DMSV (Distributed Memory – Shared Variables) and DMMP (Distributed Memory – Message Passing). Johnson's taxonomy, as claimed, fills in a category which has not been touched by Karp [100] and is more complete than that of Howe [87], through its new category GMMP which includes systems like the ELXSI 6400 [97, 132].

Furthermore, based on the buffer (address space) type, Kutti [109] classifies parallel systems into "multiprocessor" in which processors share common memory and utilize shared memory to conduct processor interactions, and "multicomputer" where each processor has its own dedicated memory; requests, processes, data and control messages are carried via the "moving buffers" [109]. According to Kutti's taxonomy, a system, in which processors/computers are linked by a network such as the internet, belongs to the "multicomputer" type. A list of existing parallel systems in each category is provided in [109].

As stated in Chapter 1, this thesis bases its discussion on parallel and distributed systems, and focuses on task scheduling issues in parallel processing as well as support for parallel programming. Therefore, this thesis assumes a homogeneous computer system in which processors are identical and fully connected with identical communication networks. However, as seen from the remaining discussion, the study

and results can be smoothly extended into heterogeneous systems.

2.2 Speedup of the Parallel System

The development of VLSI and various parallel system architectures have provided application programmers with the opportunity of pursuing an efficient and cost-effective solution to many complex application problems. The measure, *speedup*, is widely used to evaluate the performance of the parallel system. Examples of performance measurement on parallel systems can be found in [47, 50, 60, 75, 81, 83, 84, 125, 131, 147, 166].

At present, there is no standard mathematical definition for the speedup function. Researchers have proposed a number of speedup functions, based on their own experience and work. A detailed study of the speedup function is beyond the scope of this thesis. This thesis intends, through the available speedup functions, to state that the performance improvement of parallel and distributed systems is generally not as great as expected, and therefore that effort is required to realize the full potential of a parallel and distributed system.

Section 2.2.1 gives a formal definition of the speedup function. It also reviews and compares several variations of the speedup function used in the literature, with the aim of illustrating a characteristic of parallel systems, viz., non-linear speedup. Section 2.2.2 analyzes reasons why linear speedup is impossible to attain, and discusses the runtime overheads which degrade the performance of parallel systems.

2.2.1 Speedup Functions

The speedup function S of the parallel system is formally defined in [65] as:

$$S(n) = \frac{T_s}{T_p} \tag{2.1}$$

where n is the number of processors incorporated into the parallel and distributed system, T_s is the time of the most efficient sequential algorithm employed in solving a certain application problem, and T_p is the computation time of the parallel algorithm addressing the same problem.

The term *linear speedup* refers to the best performance which an n-processor parallel system can expect, i.e., $S(n) = n$ [65]. Such an ideal parallel system can produce the solution in $\frac{1}{n}th$ the time of that of an individual processor (supposing that processors here are identical, and a uniprocessor can simulate the operations of an n-processor system). However, researchers have shown that increasing the number of processors in the system does not result in a directly proportional improvement of system performance. Chu [42] proposes a "saturation effect" to demonstrate such a phenomenon in a parallel and distributed system. He points out that system performance increases significantly only for the first few additional processors that are added to a parallel system, and then begins to decrease with each newly incorporated processor (as compared to a linear increase), due to excessive interprocessor communication. That is to say, the increased overheads incurred in parallel processing somehow, to some extent, offset the benefits gained through parallelism. The saturation effect can occur when the number of processors is as few as three or four if the system or the algorithm is inappropriately designed. Examples can be found in [41, 95].

As can be seen, it is practically impossible to precisely calculate the speedup of a parallel system from Equation 2.1, since there is no guarantee that the adopted sequential algorithm is the "most efficient" one.

The formal definition of the *speedup* of the parallel and distributed system has a number of variants, which are more practical than the original definition. Such variations are summarized as follows, numbering each function with a subscript to the speedup function S. This thesis compares these speedup functions against the linear speedup function, to show that linear speedup is far beyond any parallel system.

- The speedup function S proposed by Amdahl *et.al.* [5]:

$$S_1 = \frac{1}{f + \frac{1-f}{n}}$$

  where f is the fraction of sequential operations which can not be parallelized $(0 < f < 1)$. It can be seen that $S_1 < \frac{1}{f}$. As indicated, Amdahl *et.al.* believe that the size of the sequential segments in a parallel program critically affects the performance of the parallel system.

- S as proposed by Minsky *et.al.* [122]:

$$S_2 = \log n$$

  That is to say, the speedup of the parallel system is merely limited by the log of the number of processors.

- S by Kuck *et.al.* [54]:

$$S_3 = \frac{n}{\log n}$$

This is an empirical result obtained by analyzing 86 Fortran programs. I/O and control unit timing have been neglected. It is claimed that, when $n < 10$, the performance of the parallel system is better than the linear speedup level. However, later experiments from the Parafrase system [139] on 1500 Fortran programs have proved that this function over-estimates the performance of the system, and that the speedup should be within $[\log n, \frac{n}{\log n}]$.

- S by Haynes *et.al.* [80]:

$$S_4 = \frac{1}{\frac{1}{n}P + H + Q}$$

where P and Q is the parallel and serial segment respectively, of the parallel program (obviously $P + Q = 1$); H is the overhead on each processor due to any additional steps required by the parallel algorithm. If H is ignored, i.e., $H = 0$, then this speedup function is the same as that of Amdahl's.

- S by Gustafson [76]:

$$S_5 = f + (1 - f)n$$

where f has the same meaning as that in Amdahl's speedup function. It is claimed to be a reevaluation of Amdahl's law [5]. It presents an almost linear

Speedup Function	Number of Processors					Conditions
	10	10^2	10^3	10^4	10^5	
S_1	3.57	4.81	4.98	4.99	4.99	$f = 0.2$
S_2	1	2	3	4	5	
S_3	10	50	333	2,500	20,000	
S_4	3.57	4.81	4.98	4.99	4.99	$Q = 0.2, P = 0.8, H = 0$
S_5	8.2	80.2	800.2	8,000.2	80,000.2	$f = 0.2$
Linear	10	100	1,000	10,000	100,000	

Table 2.1. Speedup of the parallel system.

function for parallel system speedup.

All the above speedup functions are derived from various researchers' experience or experimental results. They have been used to measure the performance of parallel and distributed systems. Compare these speedup functions against the linear speedup function. Table 2.1 shows the results in the case when the number of processors in the parallel system is 10, 10^2, 10^3, 10^4 and 10^5. From Table 2.1, it can be observed that the speedup of the parallel system is not a linear increase as ideally desired. The results all appear to be less than linear speedup: the more processors in the parallel system, the greater the difference from linear speedup. This is due to the runtime overhead incurred in parallel processing. Section 2.2.2 below contains a more detailed discussion.

2.2.2 Overhead in Parallel Processing

With an increase in the number of processors incorporated into the parallel and distributed system, it is expected that the directly proportional improvement in system performance can be achieved. However, from the analysis in Section 2.2.1, it is observed that linear speedup of the parallel system is impossible. This section discusses three major factors which affect the achievement of linear speedup in a parallel and distributed system.

The first factor that influences system performance is the parallel algorithm itself. Generally speaking, the algorithm partitions the whole program into a number of interrelated tasks, which can be run in parallel on processors and synchronize with each other for desired data. Even within a task, instructions can be partitioned as well to achieve further parallelism. However, in reality, parallelism does not apply to each aspect of the application program. Complete parallelism is practically unattainable. In some situations, sequential processing is required by the application problem itself. This is because there usually exists some precedence relationships between parallel tasks within the program. Consequently, parallel algorithms are typically unable to fully parallelize the computation. Therefore, even provided with sufficient processors, linear speedup of system performance is still generally impossible.

The second factor affecting system performance is the runtime overhead incurred by inter-task (or inter-processor) communication. Parallel tasks in an application program work cooperatively to solve a single problem. Subsequently, interprocessor communication (presuming the interprocess communication on the same processor can be ignored) and data synchronization is very common in parallel computation. Improper arrangement of tasks onto processors can incur excessive interprocessor communication, and thus significantly delay the execution of the program. In addition, further overhead can come from task scheduling and load balancing, if conducted dynamically at runtime. As a result, these overheads offset the advantages brought about by the parallelism.

The final factor which degrades system performance comes from hardware restrictions. When more than one task is intending to use the same resource simultaneously, competition for the shared resource occurs. Consequently, the processing of at least one task of the parallel program is delayed. Such resources may

be common main memory and shared variables, computer networks, and input/output devices etc.

In summary, linear speedup of parallel systems is unrealistic and impossible to attain on practical systems. When the number of processors increases in the system, the consequence may outweigh the advantage gained through the parallelism. Subsequently, actual system performance decreases, rather than increases as expected. This has been shown in [42].

It is desirable that increasing the number of available processors improves the whole system performance. Various strategies have been developed which allow the actual speedup to approach, as much as possible, the linear function. An appropriate task scheduling strategy (or policy) is one of the aspects which can be pursued so that the additional overhead incurred by parallelism is reduced to the minimum; this is discussed in the following sections.

2.3 Scheduling Algorithms

This section reviews current practices in task scheduling in parallel processing. Fundamentally speaking, the solution to the scheduling problem is an algorithm by which a scheduling policy can be produced so that the tasks of the parallel program are distributed onto available processors and arranged into an execution order for tasks which are assigned to the same processor. On the whole, there is the same set of constituent aspects involved in study of all scheduling algorithms. Section 2.3.1 explains in detail these aspects and their relationships. In this thesis, scheduling algorithms are divided into two broad categories: *task allocation algorithms* and *task scheduling algorithms*, reviewed in Section 2.3.2 and Section 2.3.3 respectively. Detailed surveys

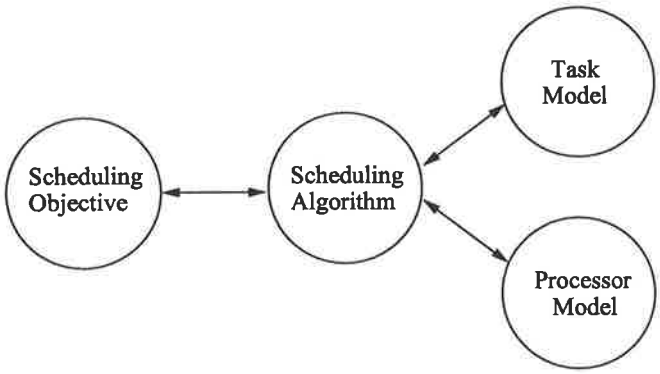


Figure 2.1. Major issues in the task scheduling problem.

can also be found in Casavant *et.al.* [35], Coffman [43], Graham *et. al.* [73], Lawler *et. al.* [114] and Norman *et.al.* [128].

2.3.1 Scheduling Aspects

As aforementioned in Section 1.1.2, the task scheduling problem is composed of four major aspects: the *scheduling objective* which is normally the performance measure to be optimized regarding the parallel system or parallel programs; the *task model* which portrays constituent tasks and possibly the precedence relationships among the tasks of a parallel program; the *processor model* which abstracts over the architecture of the underlying available parallel system on which the parallel programs can be executed; and the *scheduling algorithm* which produces a scheduling policy to distribute tasks onto the available processors.

The relationships among the major factors in the task scheduling problem is illustrated in Figure 2.1. Generally, a scheduling algorithm aims to optimize the desired performance measurement, and it determines aspects required in dealing with parallel task scheduling. The two most widely-used performance measures in scheduling research are *parallel execution time* of the parallel program (or schedule length) and

total cost of communication delay and load balance. Examples of scheduling algorithms in each category are provided in the next two sections. These two scheduling objectives result in two main streams of scheduling algorithms: task allocation algorithms and task scheduling algorithms [53], respectively. The *task allocation algorithms* concentrate on the assignment of tasks onto available processors. On the other hand, the *task scheduling algorithms* consider more factors than those of task allocation. Such factors include precedence relationships between the tasks of a parallel program (determined by the program itself), and the execution commencement order between tasks assigned to the same processor. Note that, here *task scheduling* and *task allocation* refer to different types of algorithms, while the term *scheduling algorithm* is used to refer to the general algorithm for the scheduling problem in parallel processing.

Figure 2.1 also indicates that the scheduling algorithm (as well as the scheduling objective) adopted in a parallel system determines attributes of the tasks and processors to be considered in the task and processor model, respectively. Broadly speaking, the task model is normally portrayed as a weighted undirected graph in the task allocation algorithm, since the precedence relationships are of no interest in this situation. In the task model, tasks are represented as *nodes* and weighted by computation cost, and intertask communication is represented as *edges* and weighted by communication delay. While, on the other hand, the task scheduling algorithm normally requires a weighted directed acyclic graph (DAG) to describe the task model, in which tasks are represented by *nodes* (the same as in that of the task allocation algorithm), but *edges* illustrate precedence relationships and the communication time between interconnected tasks. Therefore, the typical distinction between a task allocation algorithm and a task scheduling algorithm lies in the illustration of the task model provided to the algorithm, e.g., whether or not the precedence relationships between parallel tasks are taken into

account.

The processor model is assumed, in this thesis, to be the same when investigating both task allocation algorithms and task scheduling algorithms. The parallel and distributed system is regarded as composed of identical processors which are fully connected through identical communication networks.

In summary, the task scheduling problem can be described by the same set of scheduling aspects, which are strongly interrelated: a *scheduling objective* determines the style of a *scheduling algorithm* which itself influences the necessary attributes and structure of the *task model* and the *processor model.* However, different scheduling objectives require different methods to illustrate the task model and the processor model, and therefore results in different categories of scheduling algorithms. In the following two sections, scheduling algorithms are discussed in detail as two separate groups: task allocation algorithms and task scheduling algorithms.

2.3.2 Task Allocation Algorithms

This section examines the related work on task allocation research. Task allocation algorithms aim to minimize the total computation and communication cost of the parallel program. Precedence relationships between parallel tasks are generally neglected by these algorithms. Other objectives adopted by the task allocation algorithm, may also exist, though rare. For instance, such additional objectives include minimal system hazard [142], and smallest processor and time lower bound [58, 94, 113, 144]. They are not discussed in this section.

Task allocation algorithms usually follow traditional approaches, which are elaborated in this section: graph theory, mathematical programming and enumerative search. Other approaches utilized in this area have also been proposed, such as the

Hopfield neural network [185] and queuing theory [39, 127, 138].

The *graph theory* approach employs algorithms over the graph, such as the min-cut algorithm [78] and its improved variations [77, 99, 159], to distribute tasks onto processors, as done in various algorithms [42, 68, 161]. Other graph theories as applied to allocation algorithms can be found in [16, 19, 117, 160]. This approach can generate efficient solutions when the number of processors in the system is relatively small; otherwise it is excessively computationally intensive to obtain a scheduling policy [161].

The *mathematical programming* approach converts the task allocation problem into the pursuit of a solution to a mathematical equation with a number of constraints. It illustrates the total computation and communication cost of the parallel program (i.e., the scheduling objective) through a mathematical function as follows:

$$Cost(SP) = \sum_{p}\sum_{t} q(t,p)C(t,p) + \sum_{\forall t_1 \prec t_2} M(t_1, p_{t_1}, t_2, p_{t_2}) \tag{2.2}$$

where $Cost(SP)$ represents the total computation and communication cost of the corresponding scheduling policy SP; t_i and p_j represents a parallel task and processor respectively; $C(t,p)$ is the computation cost of task t on processor p, and $M(t_1, p_{t_1}, t_2, p_{t_2})$ is the communication cost between interrelated tasks t_1 and t_2, denoted as $t_1 \prec t_2$ (task t_2 is preceded by task t_1), when they are distributed on processors p_{t_1} and p_{t_2} respectively. The function q in Equation 2.2 is defined as:

$$q(t,p) = \begin{cases} 1 & \text{if task } t \text{ is allocated onto processor } p \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

Constraints to Equation 2.2 may include a limited memory environment which can be represented by:

$$\sum_{t,p} s(t)q(t,p) \leq R_p \tag{2.4}$$

where $s(t)$ denotes the amount of storage size required by task t, and R is the total available memory on processor p. Each constraint imposed by the application problem to the desired scheduling policy is illustrated by an equation similar to that above.

As seen, the scheduling problem is therefore transformed into a number of mathematical equations. The most commonly used method in this category is linear programming [12, 57, 140, 156, 158, 169]. Allocation algorithms adopting this mathematical programming approach include [11, 17, 18, 42, 62, 66, 118, 120, 127, 143, 170].

The *enumerative search* method generates a task allocation policy by thoroughly searching the solution space which is composed of parallel tasks, processors and constraints. This strategy forms a set of scheduling policies by considering every possible combination of the constituent tasks onto available processors. Each policy is evaluated in terms of the performance measure, and then selects the policy with the best system performance among all possible candidates. This approach guarantees the optimal solution. However, it is also obvious that this method is computationally expensive. In the case where the number of parallel tasks and processors is large, the solution space to be searched is so huge that identification of the best scheduling policy is impractical. Some algorithms have proposed a compromise and seek a sub-optimal scheduling policy. Examples include [142] and [153].

In theory, each of the three approaches stated above can obtain an optimal scheduling policy for the scheduling problem. However, an optimal scheduling solution can only be achieved under restricted situations, such as a two-processor system,

restricted task models, or a small number of tasks within an application program [51], due to the computational cost.

The allocation of parallel tasks to processors is usually undertaken prior to program execution. Thus, both the task model and the processor model are assumed to be available statically, and the pursuit of the scheduling policy does not, as a result, delay the execution time of the application program itself.

2.3.3 Task Scheduling Algorithms

This section reviews current task scheduling algorithms, which represents another stream of scheduling algorithms. The scheduling objective mostly adopted in the task scheduling algorithms is the minimization of the parallel execution time, also known as schedule length or makespan. The task model provided to the task scheduling algorithm is usually illustrated by a directed acyclic graph (DAG), as stated in Section 1.1.2.

Task scheduling algorithms can be further divided into two categories: dynamic and static, according to when they produce the scheduling policy. The dynamic approach generates the scheduling policy on the fly, i.e. as the program is executing. Algorithms presented in [7, 20, 36, 40, 107, 126, 138, 162, 178] are examples of the dynamic approach. For applications, such as those with aperiodic task arrival, and those in which task attributes can not be determined until runtime, dynamic scheduling algorithms are usually adopted to achieve good system performance as well as good load balancing. However, the dynamic approach unavoidably results in extra runtime overhead — the process of task distribution in dynamic scheduling places demands on the shared CPU resources which were originally reserved for the application tasks.

This thesis studies the static task scheduling problem, which aims to produce a scheduling policy prior to program execution. The task model and processor model are

generally assumed to be available in advance. Since it has already been proved that the task scheduling problem is NP-complete, a heuristic approach is normally adopted to achieve the sub-optimal solution within acceptable computation complexity. Two main strategies have been used to statically deal with the task scheduling problem: *cluster scheduling* and *list scheduling*. The *cluster scheduling* method first gathers user tasks into "clusters", assuming a unbounded number of identical processors are available, with the objective of minimizing the parallel time; then it assigns these clusters onto the underlying parallel architecture. Further clustering may be involved, due to the limitation of available processors. Sarkar's algorithm [150], Yang's DSC [180], and algorithms in [101, 102] are examples of cluster scheduling algorithms.

With *list scheduling*, each task is first assigned a priority value. Whenever there is an idle processor, the schedulable task with the highest priority is allocated to the available processor. On the whole, the list scheduling method considers the particular characteristics (such as processing capability) of underlying processors while distributing parallel tasks. The list scheduling approach and its variations are in widespread use, as in ERT [115], ETF [93], MH [53] and algorithms from [8, 89, 179].

Variation to the above two categories of task scheduling algorithms include the gang scheduling strategies [55, 96, 152, 171, 184] and other time and/or space sharing schemes [3, 71, 165].

Most of task scheduling algorithms presented to date focus on a deterministic task model, i.e., the task model is assumed to be precisely known prior to program execution. In addition, such a task model is further assumed not to change between program executions. Consequently, most of the present scheduling results neglect conditional branches associated with task runtime operations (task spawn, data transmission and data reception).

Consideration of conditional branches is termed *conditional task scheduling* in this thesis. This is regarded as a variation of the scheduling problem mentioned above. Little research has been undertaken in conditional task scheduling. El-Rewini *et.al.* [51] propose an algorithm which tackles this problem through simulation. The algorithm works as follows. Prior to execution, a number of simulations of possible task models (according to the execution probability of tasks) which may occur in the next execution are conducted. From these simulations, a scheduling algorithm, MH [53], is then employed to obtain a corresponding scheduling policy for each possible task model. These policies are then combined to generate a policy to distribute tasks and arrange the execution order of tasks allocated to the same processor. The algorithm also theoretically addresses the selection of representative task models, so as to reduce the computation overhead.

This thesis focuses on a static strategy to address the conditional and preemptive task scheduling problems. This problem may also be solved by adopting a dynamic approach, in which the scheduling policy is determined based on the task attributes obtained on the fly, as done in [7, 20, 40, 107, 162].

Preemptive task scheduling has been largely neglected, as compared to its counterpart non-preemptive task scheduling which assumes that data reception occurs at the beginning of the parallel task, while data transmission occurs at the end of the task. A large number of non-preemptive task scheduling strategies have been proposed, such as those mentioned above. An example of the work undertaken in preemptive scheduling can be found in [86].

This thesis focuses on the study of task scheduling algorithms, since the task allocation problem has been extensively discussed. In the remainder of the thesis, there is no distinction between the *task scheduling algorithm* and *scheduling algorithm*,

unless explicitly addressed.

2.4 Parallel Programming Support

Parallel processing has been regarded as a very promising solution to deal with increasingly complex software requirements [163]. With extensive research into the computation model, parallel algorithms and machine architectures, large demands have emerged in the translation of the computation model and parallel algorithms into operational programs. Parallel programmers desperately require support to deal with the complexity involved in parallel program development.

This section first examines general approaches of providing parallel programming support (Section 2.4.1). Section 2.4.2 illustrates several existing scheduling tools and analyzes the gap between what is required by the application parallel programmer and what is provided by current environments.

2.4.1 General Approaches

From the perspective of transparency of support for parallelism, existing support tools and environments for parallel program design and implementation can be broadly divided into three classes. The first class leaves the programmer completely unaware of the existence of parallelism, and allows them to simply develop the sequential code for the application. In this case, a special compiler, or other additional software, is required to automatically extract the parallelism and restructure the sequential program into parallel code [139]. Program analysis and transformation must be applied to the user-provided sequential code, as done in [9, 38, 59, 134, 135, 172]. Related work in the automatic extraction of parallelism from sequential programs include [48, 64, 88, 134].

The automatic extraction of parallelism from the sequential program should completely relieve the programmer of the burden of dealing with the complexity of parallel programming. However, since the compiler (or similar utility software to perform the transparent extraction) is typically designed for general purpose use, it is unlikely to achieve high efficiency for all types of application programs. Consequently, the parallelism is almost impossible to fully extract, and the efficiency of the generated parallel program is largely restricted by the algorithm employed in the sequential program which is provided by the application programmer [179].

The second category of parallel programming support takes an opposite perspective: the programmer is typically facilitated with a set of basic primitives and the parallelism is completely handled by the programmer. Such runtime libraries usually sit upon existing procedural languages, such as C or Fortran, and allow the programmer to explicitly specify the intertask communication and synchronization, as well as task generation and termination. This approach provides flexibility, efficiency, and portability to the generated parallel program. *PVM* (Parallel Virtual Machine) [69, 164] is a typical runtime library providing such programming support. Other examples include MPI [121], P4 [27], Corba [123, 129, 130, 133, 177, 181] and PORTS0 [45].

The third class of parallel programming support combines the above two extremes, i.e., parallel compiler and runtime library. High-level languages either implicitly or explicitly dealing with parallelism fall into this category. Functional programming languages do not require the explicit specification of parallelism in the program, but programmers are required to address issues such as task partitioning. Ada [21, 26], PL1 [149], Concurrent Pascal [79], and CC++ [63] are languages which present specific constructs to specify parallelism which is then realized by the compiler. An example includes the *par* construct in CC++.

Apart from high-level programming language support stated above, graphical parallel programming environments have been developed to support parallel programmers in various development phases. VPE [22], CODE [22, 174], HeNCE [15] and Linda [2, 34] belong to this kind of parallel programming support. Other environments include PTOOL [4], COIN from the INCAS project [24, 25], PEDS [183], Express [104], CAMP [141] and POKER [155]. The programmer, facilitated with a graphical parallel programming environment, typically is provided with assistance in both the design and implementation phases. In these graphical environments, the program constructs are usually similar to those in the sequential language. Therefore, program development can be undertaken in the same way as in sequential programming. An example of this is the *interface* node in VPE, which is very similar to function parameters in C. Through a graphical interface, the environment can also perform automatic code generation to further reduce the burden on the programmer [180].

Visualization is introduced into parallel programming to assist the development of parallel programs, as well as assist the programmer in understanding the parallel algorithm. Xab [14] is a tool for monitoring the execution of PVM programs. Related work is undertaken in [10, 30, 33, 31, 49, 119, 182]. A good survey with regard to the visualization of parallel systems can be found in [106, 137, 176].

2.4.2 Scheduling Tools

It is stated that the task scheduling process can be automated so as to relieve application programmers from this complex and time-consuming programming work [52]. This section reviews current scheduling support and environments provided at the application program level.

Most existing tools for scheduling support concentrate on efficient scheduling

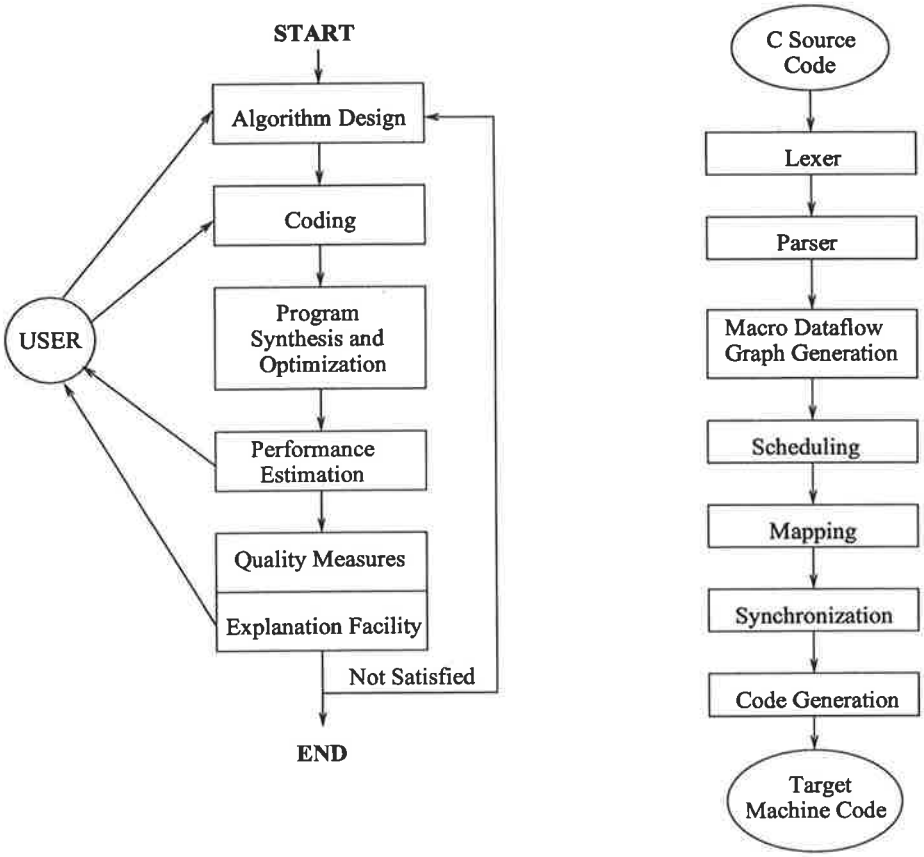


Figure 2.2. Hypertool: (a) framework and (b) program synthesis and optimization.

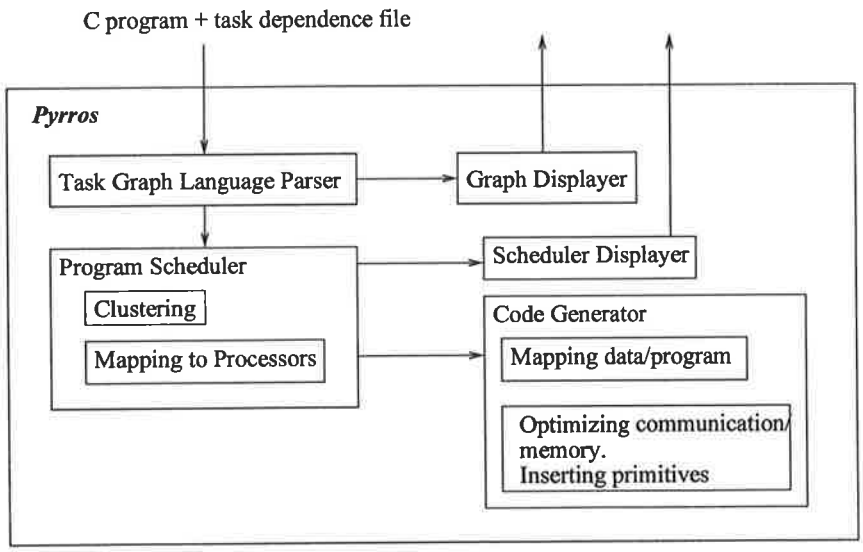


Figure 2.3. Framework of Pyrros.

algorithms to achieve good application and system performance. Hypertool [179] (Figure 2.2) is oriented toward a message-passing system, and is used throughout the whole program development cycle with comprehensive support for task scheduling. The user program (C source code) is first analyzed to generate a task model to be offered to the self-contained scheduling algorithm; new code is generated to relieve the programmer from the need to consider task scheduling issues. Another environment Pyrros [180] (Figure 2.3), developed by Yang *et.al.*, adopts a similar approach to Hypertool, but concentrates more on scheduling support and automatic code generation. Moreover, the environment Parallax [116] employs a different mechanism to tackle the scheduling problem. Parallax applies several scheduling algorithms to the same application program, and selects the most efficient scheduling policy to distribute tasks onto processors. Other research groups focus on parallel programming support by incorporating a very simple scheduling algorithm, as done in HeNCE [15] which provides integrated graphical support for parallel program development in both design and implementation phases, and a tool developed by Cai and Tuner [32] which realizes process scheduling and program visualization.

All the environments mentioned above assume that task information is known precisely prior to execution, and the incorporated scheduling algorithms do not consider any variation in the task model between executions. This assumption is only valid when the parallel tasks are utilized in exactly the same way across different program executions, and the attributes associated with tasks and parallel systems remain constant. This is termed *deterministic task scheduling.* The task information can be precisely obtained prior to a particular execution by running the program a number of times first. As stated in Section 2.3.3, conditional task scheduling algorithms have received little attention, and software tools to support conditional task scheduling are

still in demand.

Chapter 3

Conditional Task Scheduling

This chapter studies a variant of the task scheduling problem in parallel processing, termed *conditional task scheduling.*

In task scheduling research, a parallel program can be graphically illustrated by a *task model.* In this thesis, the task model illustrates *task runtime operations* between parallel tasks, which include task spawn, data transmission and data reception operations.

Fundamentally, there are two types of task models applied to the task scheduling research. One is the *deterministic task model* in which there are no conditions associated with task runtime operations, and the task model can be precisely determined prior to program execution. Such a task model does not vary between program executions, no matter how the parallel program is invoked. The other model is referred to as the *conditional task model* in this thesis. It allows task runtime operations to be guarded by conditions or placed inside loops (loops are a special case of conditional branching), hence the model is not identical across different executions. In addition, such a model is almost impossible to determine precisely prior to execution, due to the unpredictable nature of conditional branches attached to task runtime operations.

According to which task model is involved in describing the execution of a parallel program (i.e., either deterministic or conditional), the task scheduling problem, which takes the task model as input, can be consequently divided into two categories: *deterministic task scheduling* (denoted as DTS) and *conditional task scheduling* (denoted as CTS). Most current research on task scheduling, such as that reviewed in Chapter 2, is undertaken on a deterministic task model, thus the tasks and intertask communication involved in every program execution are assumed to be identical. In fact, a precise task model is a fundamental requirement in these existing algorithms. Furthermore, the analysis of the scheduling algorithm is also based on the existence of a precise task model, as conducted in [23, 37, 44, 72, 89, 92, 145]. In the analysis, the best, if not optimal, performance achieved by these deterministic scheduling algorithms takes place when the task model of the parallel program is precisely available prior to execution.

Existing deterministic scheduling algorithms and approaches can not be modified in a straightforward manner to deal with conditional task scheduling. At present, there is a lack of extensive study in the area of conditional task scheduling. An approach identified in the literature addressing the CTS problem is proposed in [51], which is reviewed in Chapter 2.

This chapter presents a strategy to tackle the conditional scheduling problem, by adaptively constructing the task model of a conditional parallel program prior to execution, and modifying task distribution across program executions. A new algorithm, named CET, is proposed to generate a scheduling policy for conditional parallel programs. Distinguishing it from current scheduling algorithms, CET considers one additional factor while scheduling the tasks of the program: namely, the *execution probability* which represents the possibility of communication between

two interconnected tasks.

The objective of conditional task scheduling adopted in this thesis is to minimize the parallel task execution time (abbreviated as PT) which refers to the completion time of the final task of the parallel program. This criteria is also known as *schedule length* and used in related work [51, 53, 151, 179, 180]. In this thesis, such an objective is achieved by producing an efficient scheduling policy statically by which to distribute tasks of the parallel program onto the underlying available processors and, at the same time, arrange the execution commencement order of tasks assigned to the same host processor.

This chapter is organized as follows. Section 3.1 formulates the processor model which describes the topology of the target machine. The conditional task model is formally defined in Section 3.2. Examples are given in Section 3.3 to illustrate the significance of conditional task scheduling and its related issues. A brief outline of the environment, named $ATME$, to deal with the conditional task scheduling problem is presented in Section 3.4. The strategy is elaborated in Section 3.5 for task model construction and Section 3.6 for the scheduling algorithm CET.

3.1 Processor Model

The topology of a parallel and distributed system is represented by a (weighted) undirected graph, in which nodes denote processors and edges represent processor connections (networks). The parallel and distributed system, PDS, is characterized by a quadruple $PDS = (P, L, \mu, \sigma)$, where:

- P: the set of processors comprising the parallel and distributed system PDS.
- L: the set of networks linking the processors in P.

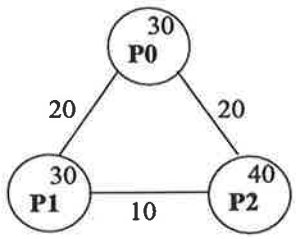


Figure 3.1. Parallel system with 3 processors.

- μ: the set representing the processing speed of each processor in P. $\mu(p)$ indicates the processing speed of a particular processor p.

- σ: the set of unit message transfer rates of each network in L. $\sigma(p_1, p_2)$ indicates the unit message transfer rate between a particular pair of processors p_1 and p_2. $\sigma(p_1, p_2) = \sigma(p_2, p_1)$.

By varying the value of $\mu(p)$ and $\sigma(p_i, p_j)$ (where p, p_i and p_j are processors), the above processor model can characterize both homogeneous and heterogeneous parallel and distributed systems. In addition, different network topologies in the system can also be represented, for instance, a fully-connected system, a local area network, a mesh or a hypercube. Processors with no direct network links can communicate via other processors.

Figure 3.1 shows an example of the processor model of a parallel and distributed system. Three processors comprise the system, each of which is associated with a value (superscript to each node) representing its processing speed $\mu(p)$. Attributes are attached to each edge between processors, representing the data transfer rate of the unit message along the network $\sigma(p_i, p_j)$.

In the study of the task scheduling problem, attributes associated with the processor model are actually determined by the adopted scheduling algorithm. For instance, some algorithms take more system characteristics into account than other algorithms.

Such system characteristics include the network contention [53] or resource availability constraints [42]. A detailed summary is found in [128].

In order to focus on the study of the task scheduling strategy, this thesis imposes a few assumptions on the processor model. It is assumed that an ideal target machine is available; that is to say, processors are assumed to be fully-connected (clique architecture) with identical communication networks. Each processor p executes one task at a time and is featured by its processing speed $\mu(p)$. The value of $\mu(p)$ for all processors is assumed to be the same. Therefore, it can be assumed that all $\mu(p) = 1$.

Communication of data between tasks which are assigned to different processors is through packets transferred over the networks. Task computation and data transmission operations can be performed almost simultaneously, i.e., the thesis assumes the existence of I/O processors. While dealing with data reception operations, the task is suspended and waits for the required data to arrive before it continues its execution. The network link between each processor pair is assigned a value, i.e., $\sigma(p_i, p_j)$, which represents the communication time per data packet. Under the assumption of identical communication networks across the system, the value of $\sigma(p_i, p_j)$ for any two processors is further assumed as 1. The communication cost between tasks scheduled on the same processor is presumed to be 0. It is also assumed that the network is communication contention free, i.e., the communication time of tasks on two processors simply depends on the data transfer rate over the network and the volume of data communicated between tasks.

With respect to the problems addressed in this thesis, the attributes of processors and networks are used to calculate computation time and communication data time of the task model (discussed in Section 3.2 below). All processor attributes are simply assumed to be 1, and the measurement of the processing speed and data transfer rate is

assumed to be identical. With these assumptions, the representation of task attribute values is largely simplified. Therefore, the thesis is not distracted by various features of a particular type of parallel and distributed system. It can then concentrate on the discussion of the strategy dealing with the conditional scheduling problem and its related issues. However, these assumptions do not affect the results and strategies delivered in this thesis and their application to heterogeneous systems. A refinement of the proposed approach in the case of heterogeneous systems is briefly discussed in Section 3.2.

3.2 Conditional Task Model

In scheduling research, a parallel and distributed program can be described by a task model which is represented by a weighted directed acyclic graph, *DAG*. Such a *DAG* representation for the parallel and distributed program is widely used [53, 151, 179, 180]. The task model is conjectured to be a critical input to the scheduling algorithm for the generation of the scheduling policy. In the *DAG*, tasks are represented by graph nodes, while task precedence relationships are represented by graph edges. Owing to the existence of conditional branches and loops within parallel tasks (especially those associated with task runtime operations), the statically-estimated attributes associated with nodes and edges of the *DAG* are unlikely to precisely represent the task model in every actual execution. This is termed a *conditional task model* in this thesis and is formally defined as $G = (T, E, C_u, C_m, T_s, T_e)$, where:

- T: the set of partitioned tasks of the parallel application program.
- E: the set of task interconnections (representing precedence relationships and communication between tasks) in the application program.

- C_u: the set of the computation times of tasks in T, i.e., the execution time required by each task, in the program, should it run. Task computation time is an important task attribute considered by most scheduling algorithms.

  Generally speaking, the task computation time is concerned with two aspects: the volume of task source code to be executed and the processing speed, $\mu(p)$, of the processor p, on which this task runs. In this thesis, it is assumed that the processing speed of all processors is identical (as stated in Section 3.1), therefore, the task computation time is regarded as simply depending on the task source code size.

  In the case of heterogeneous systems, where the processing speed and/or the architecture of processors in the parallel and distributed system is not identical, the computation time of a task can then be represented by a vector, rather than a scalar value as above. Each cell of the vector describes the computation time of the task on a particular processor. As a result, in the heterogeneous system, C_u is a $m \times n$-matrix, where m is the number of tasks and n is the number of available processors.

  Furthermore, if there exists a performance ratio between processors, the representation C_u can be simplified into the product of two matrices: one representing the computation time of each task on the selected "standard" processor, while the other represents the processing speed ratio between each processor against this "standard" processor.

- C_m: the set of communication attributes of each task interconnection (between a parent task and a child task) in E. Task models required by different scheduling algorithms are predominantly distinguished from each other by the

communication attributes used. In the conditional task model introduced in this thesis, each task communication attribute is decomposed into a pair (*communication time, execution probability*), which is further discussed below.

- The *communication time* of a pair of communicating tasks represents the time taken to transfer data between a parent and its child task, if there is such communication.

  It is assumed that the network is contention free and processors of the target machine are fully connected with identical networks. It is also assumed that the overhead involved in message formation, routing and propagation is small and can be ignored. Therefore, the communication time can be approximated by multipling the communication data size by the network data transfer rate. The size of the transferred data is measured by, for instance, the number of packets used by the communicated data. It is assumed that one unit of data is transferred in one unit of time and all networks within the system are identical (Section 3.1), hence the magnitude of data transferred between tasks is directly proportional to the time taken for communication.

  In the case of heterogeneous systems, the *communication time* between a pair of interconnected tasks, similar to the *task computation time*, can be represented by a matrix: each row and column of which represents the processors on which these two tasks may be allocated, and each cell of which represents the communication time of a pair of tasks that are resident on particular processors. Therefore, C_m is then a set of matrices, rather than a set of scalar values. This merely adds the complexity of the calculation

and representation of this communication attribute, while not affecting the result and conclusions made later in this thesis.

- The *execution probability* of a pair of interconnected tasks represents, in general, the probability that the parent task spawns, or attempts to spawn, a child task and communicate with it. This attribute portrays conditions associated with task runtime operations. A higher value of the execution probability indicates the higher probability that conditional operations are taken at runtime. It deserves to be pointed out that, for a particular program execution, the execution probability is either 1 or 0, reflecting that the operations are either taken or not taken, respectively. The execution probability attribute is introduced in this thesis, and is an important task attribute in the task model.

- T_s: the set of start tasks in the parallel program. It is assumed, with no loss of generality, that each parallel program has a unique start task.

- T_e: the set of exit tasks. Similar to T_s, it is also assumed that each parallel program has a unique exit task.

In the task model stated above, a task does not commence its execution until it is scheduled to run and it receives all input data from parent tasks. Once submitted to run, a task executes to completion without interruption, and finally transmits its output data, in parallel, to child tasks. Therefore, task execution within the study of conditional task scheduling is assumed to be non-preemptive. The problem of preemptive task scheduling is studied in Chapter 4.

Figure 3.2(a) shows an example of a conditional task model, illustrating a general case across a number of program executions. Each task is represented by a node which

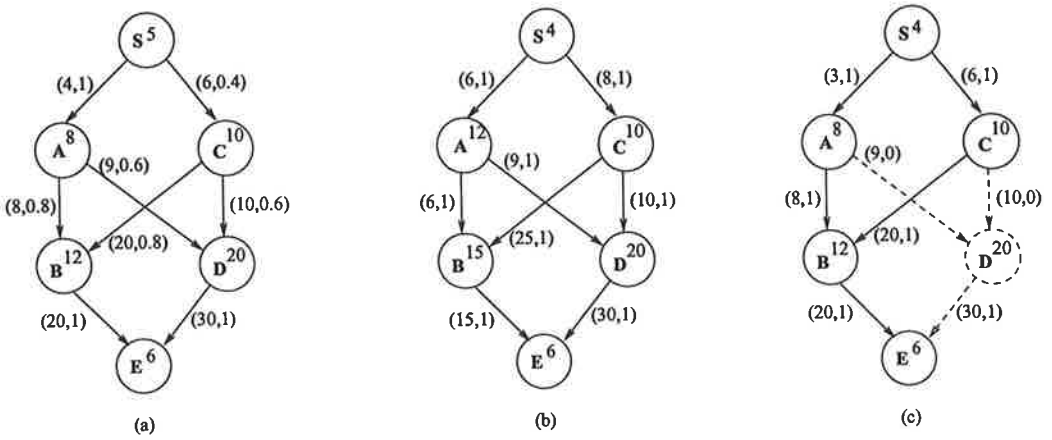


Figure 3.2. (a) Conditional task model and (b,c) potential actual task models in particular executions.

is a circle labeled with the task name. The superscript to the task name represents the computation time of that task. Task precedence relationships are represented by directed edges, to which communication attributes are attached. In Figure 3.2(a), tasks A and C depend on the successful completion of task S, but task C has a 40% probability (due to conditions guarding the communication between S and C) of executing after S completes whereas task A is always spawned by S. A task, such as C, which is not executed will have a "ripple effect" in that it cannot spawn any of its dependent child tasks such as B and D.

In a particular program execution, the actual task model is similar to those shown in Figure 3.2(b) and (c). As observed, attribute values attached to nodes and edges in the conditional task model (Figure 3.2(a)) are merely estimates and may, therefore, be different in each actual task model. Moreover, note that execution probabilities along communication edges are either 0 or 1 for a certain program execution. For the sake of clarity, when the communication between tasks has an execution probability of 1, it is represented by a solid arrow, whereas if it has an execution probability of 0, it is represented by a dashed arrow. A dashed node (such as task D in Figure 3.2(c))

describes a task which does not execute at runtime.

The conditional task model proposed here depicts the execution of the corresponding conditional parallel program in the general situation. As seen, for a certain program execution, the task model is actually a deterministic model which is similar to that in deterministic task scheduling. In this thesis, the conditional model is utilized as the foundation for the estimation of the task model in the forthcoming execution (in order to produce a scheduling policy for the conditional parallel program). Furthermore, the conditional task model, which is analyzed and summarized from program profiles, can also provide tuning suggestions for the parallel programmer to improve program design and implementation. Suggestions can be related to task partitioning, inter-task communication patterns as well as task scheduling, which is discussed in detail in Chapter 6.

The union of C_u and C_m is the *task attributes* which portray the behaviour of the tasks. This thesis introduces the concept of *usage patterns* of the parallel program, defined as the set of factors which uniquely determine the behaviour of the program (i.e, the set of task attributes). Correspondingly, the usage pattern of a parallel task (within the program) is defined as the set of factors uniquely determining the attributes of the task. In this thesis, the usage pattern (of both the program and the task) is represented by *input parameters* which are assigned values when the program (or the task) commences to execution. The value of input parameters, which may include actual parameters and global variables used by tasks, influences the behaviour of parallel tasks and determines attribute values of the tasks.

The thesis further assumes that each task has one input parameter which is assigned when the task is spawned, and represents the usage pattern of this task. As seen from the discussion below, the approach proposed in this thesis to deal with the conditional

task scheduling problem can also be applied to the situation where more than one input parameter is necessary — by introducing a more advanced regression model.

A task model, which adequately reflects task requirements (such as the time needed for task computation and communication), is critical in the production of a good scheduling policy for the efficient execution of the parallel tasks. Most current scheduling research assumes *a priori* knowledge of the task model. Nevertheless, in reality, it is generally impossible to achieve a precise task model for a conditional parallel program prior to execution.

In the following sections, two examples are presented to illustrate two typical issues relevant to the conditional task scheduling problem. Then, this thesis describes its approach to deal with the CTS problem, through a preliminary framework (which is further elaborated in Chapter 6). Detailed discussion of the strategy applied to the CTS problem is delivered in Sections 3.5 and 3.6.

3.3 Examples of Conditional Task Scheduling

In this section, an example of a parallel program is used to illustrate the significance of two important issues in conditional task scheduling: (a) that the scheduling policy should be adjusted between consecutive program executions (i.e., there should be different policies at different times when the program is invoked and executed), and (b) that the task model should be adequately (not necessarily precisely) estimated prior to program execution, in order to achieve satisfactory system and program performance.

The parallel program is represented by its conditional task model as in Figure 3.2(a), while Figure 3.2(b) and (c) show two corresponding actual task models in two program executions. The scheduling algorithm used is CET (elaborated in Section 3.6). The

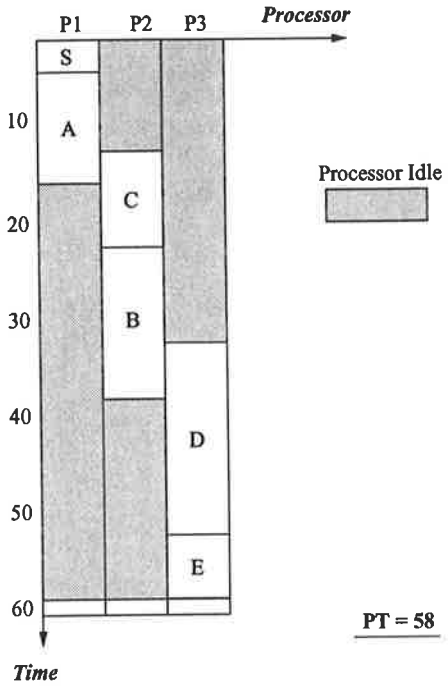


Figure 3.3. Performance of Figure 3.2(b).

scheduling policy is represented by a chart with two axes, as shown in Figure 3.3. The entire task scheduling process follows that stated in Section 1.1.2. The processor model is discussed in Section 3.1, which states that the underlying system is assumed to be a fully-connected network of homogeneous processors. The performance of the application program is measured by *parallel execution time* (PT).

3.3.1 Adjusting Scheduling Policy Between Executions

Unlike the scheduling policy for a parallel program in deterministic scheduling which is an invariant across all executions of a program, the scheduling policy in conditional task scheduling should be adjusted for each program execution, according to the variation in the task model, in order to achieve improved performance.

In this example, suppose there are two sequential executions of a program, which follow different actual task models. In the first execution, presume that the actual task

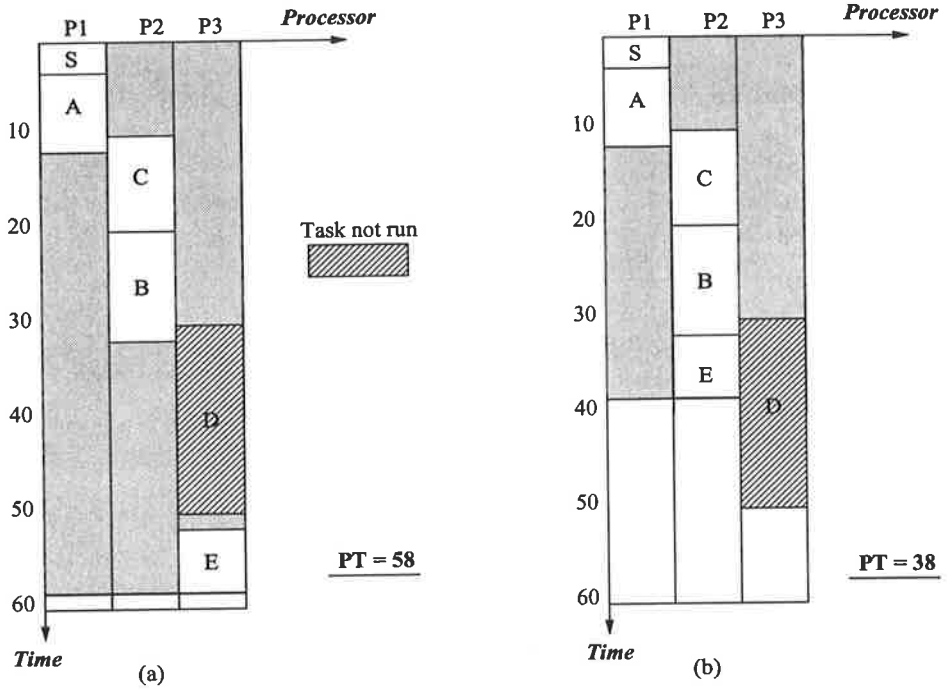


Figure 3.4. Performance of Figure 3.2(c) when (a) considering and (b) not considering the variation of the task models in different executions.

model is accurately estimated as shown in Figure 3.2(b), i.e., all tasks and intertask communications are executed. Consequently, the corresponding scheduling policy is shown in Figure 3.3. The performance has $PT = 58$.

The second program execution follows the task model as shown in Figure 3.2(c) in which task D does not execute. If presuming that the scheduling policy in the second execution remains unchanged, then the performance of the second execution, under the same scheduling policy as the first one's, will be $PT = 58$, shown in Figure 3.4(a). Now if the task model in the second execution can be predicted, then a new scheduling policy can be generated and performance of $PT = 38$ can be achieved, as displayed in Figure 3.4(b). As observed, in this example, the performance improves by 34% (i.e., $(58 - 38)/58$) by adjusting the scheduling policy to correspond to the change in the task model.

It is true that the variation in the model cannot be precisely known *a priori*;

however, it is possible that the model can be predicted with adequate accuracy, based on past execution profiles. Nevertheless, the variation of the task model between executions should not be neglected and the scheduling policy should be adjusted across a series of program executions.

3.3.2 Significance of an Accurate Task Model

In deterministic task scheduling research, it is typically assumed that a precise task model is available prior to program execution. However, owing to the presence of conditional branches in tasks of parallel programs, this assumption is not valid for the study of conditional task scheduling. Estimating attribute values of the task model is a significant step in establishing the task model for a parallel program. This section shows the influence of an accurate task model on the scheduling policy, and hence to the system and program performance.

Consider two task models which are predicted prior to program execution, one of which is precisely the model executed at runtime while the other is not. The two estimated task models are provided to the scheduling algorithm to obtain a task distribution and determine system performance, PT, for each distribution. Assume that the inaccurate task model of Figure 3.2(c) is predicted prior to program execution, Figure 3.5(a) shows the corresponding scheduling policy (for the time being, ignore the shaded part representing task B which reflects the actual situation at runtime). At runtime, however, task D is actually executed while task B is not, that is to say, the predicted task model is inaccurate. The performance under this scheduling policy is $PT = 86$, as shown in Figure 3.5(a).

Consider now the accurate task model and its influence on system performance. With the accurately-estimated task model (that is, the task model which predicts that

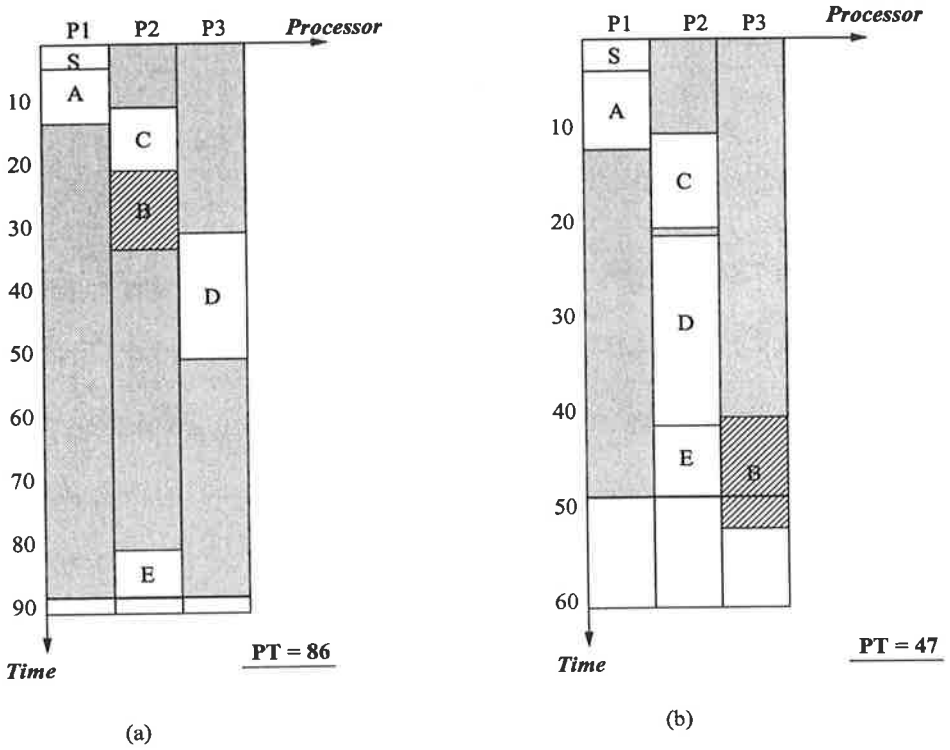


Figure 3.5. Performance difference when execution probabilities are not accurately estimated.

task D runs and task B does not), the scheduling policy of Figure 3.5(b) is achieved, and the system performance is $PT = 47$. As seen, in this case, a 45% (i.e., $(86 - 47)/86$) performance improvement is achieved by providing an accurate task model to the scheduling algorithm.

Examples such as those above indicate that strategies, which can achieve high efficiency in solving the deterministic task scheduling problem, may not provide the same performance efficiency for conditional task scheduling. It can be seen that conditional task scheduling must, by its nature, consider more factors than its deterministic counterpart. The scheduling policy should be adjusted between program executions according to variations in the task model. Furthermore, accurate prediction of the task model can assist in producing a good scheduling policy which helps approximate optimal system performance. Research effort is needed to propose new

approaches to tackle the conditional task scheduling problem.

3.4 Strategy for CTS: *ATME*

Various heuristic approaches have been proposed to efficiently deal with task scheduling within acceptable computation complexity. However, it is difficult to isolate one heuristic which achieves better performance than any others in all cases. Little attention has been paid to conditional task scheduling until recently. Heuristics need to be determined to fill this gap.

This thesis studies the conditional scheduling problem from two perspectives: the scheduling algorithm and the task model. This section presents a preliminary framework of *ATME*, an environment aimed at dealing with the conditional and preemptive parallel programming.

Figure 3.6 briefly describes the major functional components which comprise *ATME*. The *pre-execution processing* component of *ATME* receives partitioned parallel tasks of a (conditional) parallel program, as well as the information regarding the underlying parallel and distributed architecture. This component analyzes parallel tasks and produces instrumented tasks for the purpose of generating runtime data related to parallel tasks and task communication patterns. The *pre-execution processing* component also produces the *processor model* for the target machine to be used in task distribution.

The strategy for the conditional task scheduling (*CTS*) problem is implemented through two steps in *ATME*. The first step involves the construction of the task model for the forthcoming execution on the basis of past program execution profiles and model estimates. This is realized by the *task model construction* component. Once a task

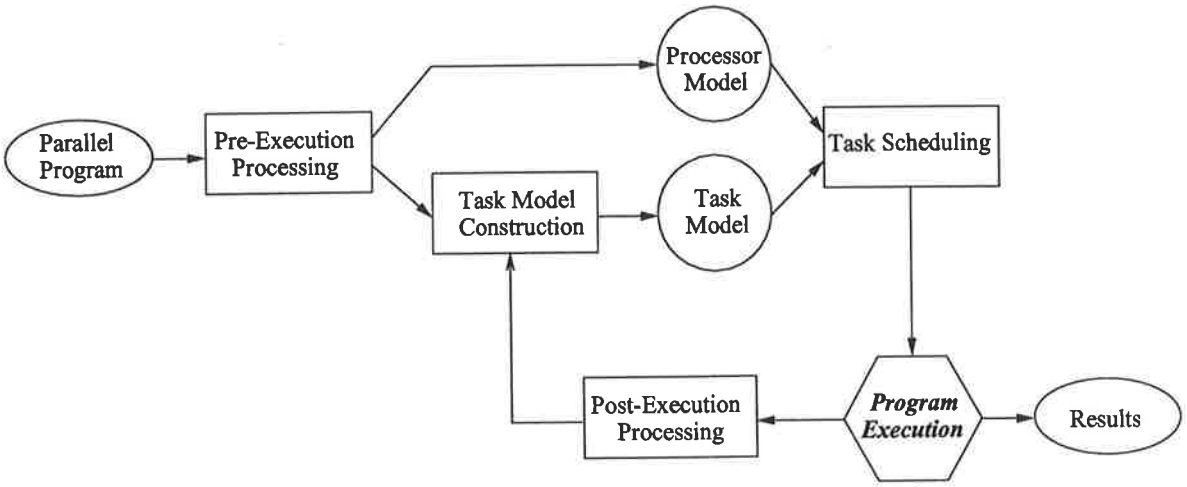


Figure 3.6. ATME framework (outline).

model is established, the second step then provides this estimated model to a scheduling algorithm, named CET, to produce a scheduling policy. This is implemented in the *task scheduling* component.

The strategy stated here works efficiently in the case when usage patterns of the parallel program are relatively stable, i.e., input parameters provided for each task when it is spawned do not change radically between consecutive executions, in which case the task model can then be estimated beforehand with reasonably high accuracy. This is strongly supported by the experiments presented in Section 7.2.5.

In the remaining sections of this chapter, effort is made to practically deal with the conditional task scheduling problem. Sections 3.5 and 3.6 discuss in detail the two main steps of the $ATME$ strategy. The design and technical details of the developed environment $ATME$ is presented in Chapter 6. Experimental results regarding the performance and features of the proposed strategy are given in Chapter 7.

The *post-execution processing* component of $ATME$ collects all runtime-generated data regarding the program as well as the distributed system as a whole, and retains them as program profiles, which acts as input to the *task model construction* for the

estimation of the task model in the forthcoming program execution.

3.5 Task Model Construction

With respect of conditional task scheduling, the task model consists of the following components: *task interconnection structure*, and the task attributes which refers to the *task computation time*, *communication time* and *execution probability* between tasks. The task interconnection structure contains all the possible tasks that compose a parallel program and all the possible interconnections between the tasks. That is to say, it describes task precedence relationships in the program when all conditional branches associated with task communication are assumed to be taken. The task interconnection structure can be established through static program analysis (discussed in Chapter 6). Therefore, this section only studies task attribute estimation.

Preparatory work must be undertaken prior to program execution in order to profile the parallel program. In this thesis, static program analysis (in addition to task instrumentation) is performed on the user source code (discussed in Section 6.4), to generate task runtime data. This is handled by the *pre-execution processing* component in Figure 3.6. The runtime information to be captured is related to the computation time of each task, communication time between any two interconnected tasks, execution probability (either 1 or 0 at runtime) between communicating tasks. For each application, if there is a factor which is known beforehand that determines task attribute values, i.e., the *input parameter*, then this fator is captured too, for use as a regressor to predict task attributes.

The prediction of task attributes is based on what is captured during previous executions — a method which is commonly adopted in predicting runtime

attributes [154]. The number of past executions retained is specified by the user. Two techniques are utilized in the construction of the task model (specifically, for the estimation of task attributes): one is the *linear regression model* to estimate continuous task attributes including computation time and communication time of tasks, while the other is based on a *finite state machine* to predict discrete task attributes, such as the execution probability.

Section 3.5.1 explains the linear regression technique adopted to undertake the prediction for task attributes, followed by a discussion on estimating task computation time and communication time in Section 3.5.2. The estimation of task execution probability is examined in Section 3.5.3.

3.5.1 Linear Regression Model

This section describes the process and results of the linear regression model, by which the *task model construction* component in Figure 3.6 can predict the value of task computation time and task communication time of a conditional parallel program. Linear regression has been extensively studied in the mathematical and statistical area [46, 74, 103, 157, 173]. It has been widely adopted in many applications [82].

Suppose there are n executions, each of which captures two data values x and y. The value of x and y in the i^{th} execution is denoted as x_i and y_i respectively. The objective is to predict the value of the sample y in the $(n+1)^{th}$ execution, based on the x and y values collected in the past n executions and a given value of x_{n+1}.

The linear regression model is represented as follows.

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

where α, β are regression coefficients and ε_i is the error term (shock or disturbance) which can usually be neglected. y represents the data whose value is to be predicted, and x is called the *regressor* in the regression model. It is assumed that the value of y can be determined through that of x. In the situation where there is more than one sample involved in deciding the value of y, a more advanced regression model is required [74, 103], which is beyond the scope of this thesis.

The linear regression aims to determine the value of α and β, based on the collected values of x and y in the past n executions. Therefore, in the $(n+1)^{th}$ execution, with the availability of the value of x_{n+1} and calculated values of α and β, the forthcoming value of y_{n+1} can be predicted as close as possible to the actual value.

Parameters α and β can be achieved by the least squares method [148]. Let

$$f(\alpha, \beta) = \sum_{i=1}^{n} [y_i - (\alpha + \beta x_i)]^2$$

The value of α, and β must be determined to minimize the function $f(\alpha, \beta)$. Thus, solving the equations:

$$\begin{cases} \frac{\partial f}{\partial \alpha} = -2 \sum_{i=1}^{n} [y_i - (\alpha + \beta x_i)] = 0 \\ \\ \frac{\partial f}{\partial \beta} = -2 \sum_{i=1}^{n} [y_i - (\alpha + \beta x_i)] x_i = 0 \end{cases}$$

it is determined that:

$$\begin{cases} \alpha = \frac{\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i x_i - \sum_{i=1}^{n} y_i \sum_{i=1}^{n} {x_i}^2}{\left(\sum_{i=1}^{n} x_i\right)^2 - n \sum_{i=1}^{n} {x_i}^2} \\ \\ \beta = \frac{\sum_{i=1}^{n} y_i \sum_{i=1}^{n} x_i - n \sum_{i=1}^{n} y_i x_i}{\left(\sum_{i=1}^{n} x_i\right)^2 - n \sum_{i=1}^{n} {x_i}^2} \end{cases}$$

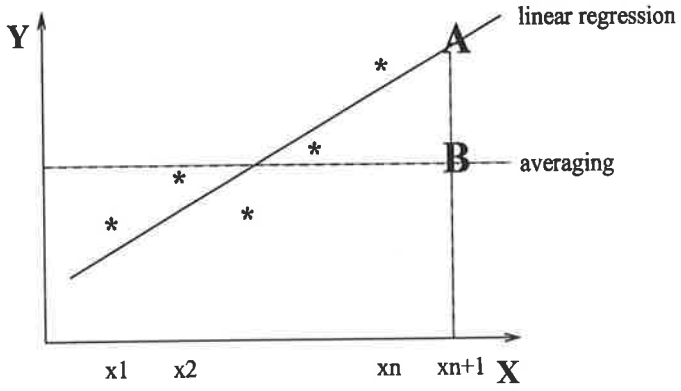


Figure 3.7. Predicted value under linear regression model against averaging model.

Therefore with the availability of α and β, as well as the value of x_{n+1}, y_{n+1} can then be obtained by:

$$y_{n+1} = \alpha + \beta x_{n+1}.$$

The linear regression model introduced above is suitable for the prediction of the future value of a variable which changes continuously (as compared to a discrete variable). Therefore, the linear regression model is adopted in this thesis to estimate the task computation time and the communication time between tasks. Section 3.5.2 below provides a detailed discussion.

As seen, the regression model takes consideration of the tendency of the past experiment history. Figure 3.7 graphically illustrates how the model works. In the past n experiments, sample values (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), are collected. With this data, a line of best fit (the "solid line" in the figure) is drawn to best approximate these n-pairs of values. With a given regressor value, x_{n+1}, the corresponding value on the line, i.e., y_{n+1}, can then be estimated (marked by letter **A**). This method considers the tends visible in the previous recorded values. On the other hand, the averaging model estimates y_{n+1} according to the average value of y_1, y_2, ..., y_n. In Figure 3.7,

this y_{n+1} value from the averaging model is marked by letter **B**. It is clear that the averaging method ignores the relationship between x_i and y_i, and therefore fails to consider any trends apparant in the historic data.

Section 7.2.5 demonstrates the experiment results regarding the accuracy of the linear regression model.

3.5.2 Estimation of Task Computation and Communication Time

Task computation and communication attributes of the task model in the $(n+1)^{th}$ execution are estimated by analysis of corresponding data values collected in the previous n executions (the exact value of n is user-defined). The input parameters of each task are also assumed to be captured in the previous n executions.

In the situation when attributes of a task is determined by the task's *input parameter*, such a parameter can be used as the regressor in the regression model (shown in Section 3.5.1). Consequently, the estimation of task computation time and communication time is divided into two steps: first the value of the task *input parameter* is estimated; and then the task attribute is predicted. In both steps, the estimation is undertaken by the linear regression technique.

In the first step the value of the task's *input parameter* in the $(n+1)^{th}$ execution is estimated (that is, denoted by y). This thesis takes the task *execution number* as the regressor x in the regression model. For instance, in the first execution, the value of x_1 is 1, in the second execution, x_2 is 2, and so on. In each execution, the *input parameter* is captured and recorded. With the regression model and least squares method, the value of α and β is then calculated. In the $(n+1)^{th}$ execution, with the regressor x_{n+1}

as $(n+1)$, and with the value of α and β which is calculated according to the formula stated in Section 3.5.1, the value of the *input parameter* of a task in the $(n+1)^{th}$ execution is then achieved. This process is generally described in Section 3.5.1.

In the second step of predicting the task attribute, the *input parameter* value acts as the regressor x, while the attribute to be predicted (i.e., either the task computation time or task communication time) is y in the regression model. A new set of α and β values is then determined. Since the *input parameter* of the task in the $(n+1)^{th}$ execution is available in the first step, with the equation:

$$y_{n+1} = \alpha + \beta x_{n+1}$$

the value of the task attribute in the forthcoming $(n+1)^{th}$ execution is obtained.

In the circumstance when the *input parameter* of tasks is not explicitly stated or is difficult to analyze, the two steps stated above are combined into one. The attributes to be predicted, i.e., y in the regression model, is still either the task computation time or the task communication time. The regressor, x, utilizes the *execution number*, illustrated as in the first step above. That is, the estimation of the *input parameter* is skipped. In this way, the estimation strategy avoids the complexity of determining the input parameters of tasks or the parallel program.

3.5.3 Estimation of Task Execution Probability

Actual task models collected in each user's execution profile are employed to predict the task model in subsequent executions by that user. Each interconnection in the *actual* task model is labeled with either 1 or 0, representing the actual value of the task attribute, *execution probability*. This task attribute indicates whether task spawn and

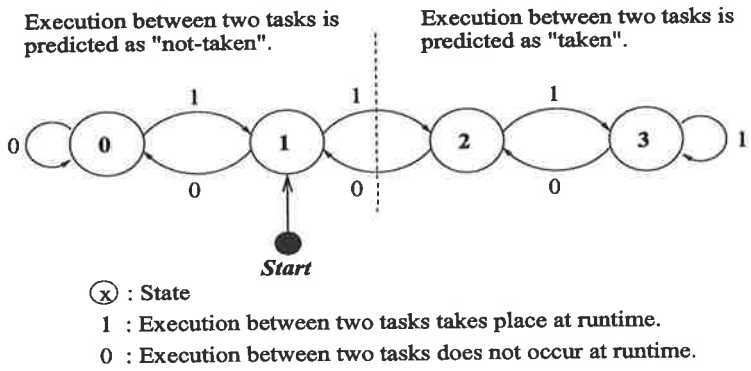


Figure 3.8. 4-state finite state machine to predict execution probability.

communication along that route occurs or not at runtime (1 for occurrence).

In this thesis, the execution probability of an interconnection in the task model is predicted by applying the corresponding values captured in previous executions into a m-state finite state machine (FSM). Figure 3.8 provides an example of such a FSM, where $m = 4$. Starting at the start state (state 1 in this case), the FSM is navigated using the execution probability values for that interconnection from n previous actual task models (in the time order of program execution). For instance, in the first execution, the 0 value of the recorded execution probability (of that interconnection) brings the state back to State 0; while the 1 value moves the current state to state 2. The FSM eventually reaches a state when the last recorded value of that interconnection is applied. Therefore, a "01110" recorded sequence of a certain task interconnection in the past five executions finishes at state 2 in the FSM.

There is a threshold state in the FSM which is used to predict whether the execution path along the interconnection will, or will not, take place. In Figure 3.8, if the number of the final state is equal to or greater than that of the threshold state (State 2), then the execution path between this pair of tasks is predicted to be "taken", otherwise it is assumed to be "not-taken".

The prediction accuracy of the execution probability depends on the number of

states, the initial and threshold states in the FSM, as well as the relative stability of the usage patterns of the application and the number of values retained in the execution history.

In the above Sections 3.5.2 and 3.5.3, the estimation of task attributes requires initial data stored in program databases (Figure 3.6) so as to provide a basis for task model construction in subsequent executions. This can be achieved through a small number of initial runs which requires no knowledge of the task model.

In general, the task model of a conditional parallel program is *conditional*. However, when parallel tasks are distributed onto the underlying processors, the task model is regarded as *determined*, since task attributes in the model have all been estimated and have concrete, rather than statistical, values.

Based on the predicted task model for the forthcoming program execution, the next step in the strategy to tackle conditional task scheduling is an algorithm to realize task distribution among underlying available processors. This is elaborated in Section 3.6.

3.6 Conditional Task Scheduling Algorithm

This section proposes an algorithm, named CET, for the purpose of producing a scheduling policy by which tasks of the application program can be allocated onto the underlying available processors and to define the execution order of tasks assigned to the same processor. CET is different from existing scheduling algorithms in two respects:

- CET takes more factors of the task model into consideration: not just task attributes such as task computation and communication time, but also the execution probability which portrays the possibility of communication between

interconnected tasks, due to the existence of conditions associated with task runtime operations.

- CET is provided with a task model which is estimated each time the program is executed. This is based on the fact that a scheduling policy which is efficient for one execution may not achieve the same performance for the other executions in conditional task scheduling.

Section 3.6.1 summarizes the notation used in the scheduling algorithm, CET, which is discussed in Section 3.6.2.

3.6.1 Notation

Before formally introducing the scheduling algorithm CET, the notation adopted in CET is presented and summarized as follows.

- T: the set of all tasks in the application; t: a specific user task.
- P: the set of all available processors in the system; p: a specific processor.
- $\prec$: the precedence relationship between a pair of directly-related tasks in the task model. $t_1 \prec t_2$ indicates that task t_1 potentially spawns task t_2.
- $\| S \|$: the number of elements in set S.
- W: the set of unscheduled tasks which have no parent tasks or whose parent tasks have already been allocated to processors. That is to say, the tasks in W are schedulable.
- W_e: a subset of W, in which tasks are predicted to run by the *task model construction* component.

- W_n: a subset of W, in which tasks are predicted not to run. Initially, $W_e \bigcup W_n = W$.
- $U(t,p)$: the computation time of task t on the processor p.
- $F(t,p)$: the execution finish time of task t on the processor p.
- $M(t_1,t_2)$: the data transmission time between task t_1 and t_2, which are resident on the processors p_1 and p_2 respectively. If $p_1 = p_2$, then $M(t_1,t_2)$ is assumed to be 0; If the execution probability between t_1 and t_2 is estimated as 0, then $M(t_1,t_2)$ is also regarded to be 0.
- $R(t_1,t_2)$: the execution probability between task t_1 and t_2.
- $G(t)$: a function returning two values related to the distribution of task t: the allocated processor p and the execution order of task t on p, respectively.
- $D(t)$: the processor onto which task t is distributed.
- $N(p)$: the current tasks allocated to processor p.
- $A(p)$: while tasks are being scheduled, $A(p)$ represents the earliest time when the processor p is available to execute a new task.
- $Q(t)$: the number of unscheduled parent tasks of task t.
- $S_0(t,p)$: Among all the schedulable tasks and the available processors, task t has the smallest "earliest start time", and its resident processor is p.
- $S_1(t,p)$: the earliest possible start time of task t on processor p, ignoring the earliest available time of p (i.e., $A(p)$).

- $S_2(t,p)$: the earliest start time of task t on processor p, considering p's earliest available time.

- Y: a set of tasks with the same earliest start time.

- $colevel(t)$: the co-level number of task t in the task model. The co-level of a task is defined as the length of the longest path from the start task to the task t in the DAG. The path length is counted by the number of vertices passed.

3.6.2 Scheduling Algorithm CET

This section presents the scheduling algorithm developed in this thesis, CET (Conditional Earliest Task), to tackle the conditional task scheduling problem. The CET algorithm is given as follows:

1. Initialization:

   $\forall t_i \in T, Q(t_i) =$ the number of parent tasks of t_i

   $\forall p_j \in P, A(p_j) = 0$

   $\forall t_i \in T, G(t_i) = (Null, Null)$

2. Select a task to be scheduled and its host processor, denoted as t_k and p_l respectively: Let W be the set of tasks, each of which has not been scheduled yet or whose parent tasks have all been allocated:

   $$W = \{t \mid t \in T, Q(t) = 0, G(t) = (Null, Null)\}$$

   If $\parallel W \parallel = 0$, then the task scheduling procedure completes; otherwise, split W into two subsets W_e and W_n:

$$W_e = \{t \mid t \in W, \exists t_1 \in T, t_1 \prec t, R(t_1, t) = 1\}$$
$$W_n = W - W_e$$

(a) If $\| W_e \| > 0$, select a task in W_e and a processor in P with the "earliest start time": for $\forall t_i \in W_e$, and $\forall p_j \in P$, assume task t_i is allocated onto processor p_j, and calculate:

$$\begin{aligned} S_1(t_i, p_j) &= \max\{(F(t_{i1}, p_{j1}) + M(t_{i1}, t_i) * R(t_{i1}, t_i)) \mid \\ &\qquad \forall t_{i1} \prec t_i, D(t_{i1}) = p_{j1}, D(t_i) = p_j\} \\ S_2(t_i, p_j) &= \max\{S_1(t_i, p_j), A(p_j)\} \end{aligned}$$

where, if the execution probability between task t_{i1} and t_i is predicted to be 0, i.e, $R(t_{i1}, t_i) = 0$, then the expected communication time between these two tasks is regarded as 0; if the two tasks t_{i1} and t_i are allocated onto the same processor, then $M(t_{i1}, t_i) = 0$; if t_{i1} is predicted not to be executed at runtime, then $F(t_{i1}, p_{j1}) + M(t_{i1}, t_i) = 0$.

Evaluate S_2 among all schedulable or free tasks (i.e., in the set W_e) on all available processors:

$$S_0(t_k, p_l) = \min\{S_2(t_i, p_j) \mid \forall t_i \in W_e, \forall p_j \in P\}$$

t_k is then selected as the next to be scheduled task, if it is the unique task which has "earliest start time" (i.e., S_0).

Let Y represent the set of tasks with an identical "earliest start time":

$$Y = \{t_i \mid t_i \in W_e, S_0(t_i, D(t_i)) = S_0(t_k, D(t_k))\}$$

If $\| Y \| > 1$, choose a task t_k with the minimum value of $F(t_k, D(t_k))$, and let $p_l = D(t_k)$. If more than one task has this minimum value, then randomly select one.

(b) If $\| W_e \| = 0$, select a task from the non-executed task set W_n by its co-level number: for $\forall t_i \in W_n$, and $\forall p_j \in P$, choose a task with the minimum *colevel* value [1] as:

$$colevel(t_k) = \min\{colevel(t_i) \mid \forall t_i \in W_n\}.$$

Then, assuming the execution between t_k and all its parent tasks occurs, select the allocating processor p_l by the earliest start time stated above.

In this step, regardless of which of the above two cases occurs, the processor p_l is chosen on which the selected task t_k is to be distributed.

3. Update the state variables:

$N(p_l) = N(p_l) + 1$

$G(t_k) = (p_l, N(p_l))$

$A(p_l) = A(p_l) + U(t_k, p_l)$

$W = W - \{t_k\}$

For any child task t_i of task t_k, $Q(t_i) = Q(t_i) - 1$. If $Q(t_i) = 0$, then $W = W \bigcup \{t_i\}$.

4. Go to step 2.

From the above algorithm, it can be observed that, at any moment, each *schedulable* task, or *free* task, (that is, a task such that all its parent tasks have been distributed onto processors) in the application program is assigned a priority value as its "earliest

start time". The free task set W is divided into two subsets: W_e in which tasks are expected to execute and W_n in which tasks are not expected to execute. Only when W_e is empty, is there a task in W_n to be distributed. In the end, all user tasks, including those predicted not to run, are scheduled onto the underlying processors and arranged in the order of execution.

CET distributes tasks in two steps. The first step is simply that for any task which is ready to be scheduled and predicted to run in the task model construction, CET calculates its S_2 for all available processors. The processor on which the lowest value of S_2 is obtained and its corresponding task are then selected. In the second step, among all schedulable tasks, CET selects the one with the smallest value of S_2, and distributes this task onto the processor which has been chosen in the first step.

With regard to those tasks which are not expected to execute at runtime, the *colevel* measure is adopted to select the task and determine a processor for distribution. The *colevel* is widely used in constructing the priority table in list scheduling [1, 53]. In this way, all tasks are individually scheduled onto the underlying available parallel system.

In the CET algorithm, the major job undertaken is to, at a certain point of time, select an available processor for a schedulable task. This is realized through calculating and comparing either the *earliest start time* of all schedulable tasks among all available processors, or the *colevel* of all schedulable tasks (predicted not to run). The upper time limit of such a procedure is $O(mn)$ where n is the number of tasks of a parallel program and m is the number of processors in the system. This procedure is repeated whenever the distribution is conducted for a parallel task. Therefore, the overall time complexity of the algorithm CET is $O(mn^2)$.

As seen, the algorithm CET follows the list scheduling approach, which has been adopted by a number of existing scheduling algorithms to distribute parallel tasks [90].

This thesis does not focus on the "most efficient" algorithm to deal with the conditional task scheduling problem. What this thesis contributes lies in the strategy to tackle the conditional scheduling problem, and moreover, the practicality of the strategy. The strategy is designed with a distinct interface between the task model construction and the scheduling algorithm. Therefore, future results and development in any part of the strategy can be easily incorporated in so that the parallel computation can be enhanced immediately.

Chapter 4

Preemptive Task Execution and Scheduling

In deterministic non-preemptive parallel applications, the task model is assumed to be precisely known prior to program execution. As mentioned in Section 3.2, at runtime, a task does not commence its execution until it receives all its required data from its parent tasks. It then executes to completion, and transmits data to its succeeding child tasks. Therefore, communication between tasks are restricted to the beginning and the end of each task. However, in reality, task spawn and message-passing between tasks do not necessarily only occur at the start and end of a task.

This chapter concentrates on the study of "preemptive task execution and scheduling", and related issues. In this thesis, preemption refers to the fact that the execution of a task is interrupted by another task and resumes later on, thus the commencement or completion time is altered as compared to that would otherwise occur in the non-preemptive situation. The preemption is regarded as the consequence of permitting task runtime operations to occur at any place within the task. The outcome of task preemption is that the child task may commence its execution before

the completion of its preceding parent tasks. The motivation of the task preemption study originates from the intuitive belief that the earlier a task (within a parallel program) commences its execution, the earlier its child task(s) can start to run, and therefore, the higher system performance will be. This belief is supported by experimental results presented in Chapter 7.

The study in this chapter includes two aspects: *preemptive task execution* (abbreviated as PTE) and *preemptive task scheduling* (PTS). Preemptive task execution investigates the performance improvement due to the occurrence of preemption at runtime. All tasks have already been scheduled onto the underlying available processors (by a certain scheduling policy). Such preemption can take place on either the same or different processors. Preemptive execution may either improve or degrade system performance. This chapter proposes an approach by which system performance improvement can be guaranteed. On the other hand, preemptive task scheduling concentrates on not only the scheduling algorithm itself, but also on issues which relate to that algorithm, such as the task model. This chapter introduces a "preemptive task model" to illustrate preemption between tasks. It also presents a new scheduling algorithm, named PET, to deal with the preemptive task model.

In this chapter, the underlying parallel and distributed system is captured by the processor model, as stated in Section 3.1. That is to say, the target machine is regarded to have the identical processors fully connected through identical networks. The scheduling objective is the same as that in Chapter 3, i.e., the parallel execution time PT.

Parallel program execution is assumed to be "deterministic"; that is, it is assumed that no conditions are associated with task runtime operations (task spawn, data transmission and data reception). The behaviour of the parallel program is the same

across different program executions. Consequently, the task model can be determined precisely prior to execution by running the program a number of times. Chapter 6 addresses the case which mixes the preemption and conditional branches in parallel programming. These assumptions simplify the task model in the study of preemptive execution and scheduling in this chapter so as to allow its properties to be examined and compared to non-preemptive scheduling.

This chapter is organized as follows. Section 4.1 illustrates the preemptive task model. An example of preemptive task execution is presented in Section 4.2. Preemptive task execution is discussed in Sections 4.3. A preemptive task scheduling algorithm, PET, is presented in Section 4.4.

4.1 Preemptive Task Model

The task model of a preemptive parallel program is represented by a weighted directed acyclic graph as shown in Figure 4.1. It can be observed that the preemptive task model is similar to the conditional task model which is presented in Section 3.2. The distinction lies in different task attributes applied to the task model, required by different scheduling algorithms. The preemptive task model can be formally defined as $G = (T, E, C_u, C_m, T_s, T_e)$, where:

- T: the set of tasks in the parallel program.
- E: the set of task interconnections in the program.
- C_u: the set consisting of the computation time of each task in T. A detailed explanation is found in Section 3.2. The discussion of task computation time regarding the heterogeneous system is also undertaken in Section 3.2.

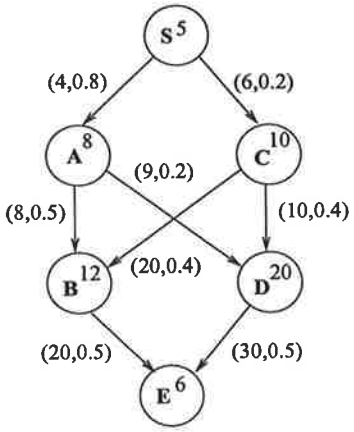


Figure 4.1. The preemptive task model.

- C_m: the set of communication attributes of each task interconnection (between a parent task and a child task) in E. Each communication attribute is a pair (*communication time, preemption start point*), discussed as follows:

  - The *communication time* represents the total time taken to transfer data between a parent and its child task. Refer to Section 3.2 for details.

  - The *preemption start point* represents the point at which message transmission within a parent task to a child task may first commence at runtime. It is defined as a ratio given by the following formula:

$$V(m,c) = \frac{CT(c) - CT(m)}{U(m)}.$$

    where:

    * m, c: a parent task, m, and its child task, c.

    * $CT(t)$: the execution commencement time of task t.

    * $U(t)$: the computation time of task t.

* $V(m, c)$: preemption start point at which messages within a (parent) task m are transmitted to its child task c. The extreme values of such a preemption start point occurs at two locations. One takes place at the beginning of the parent task m, i.e., the child task, c, commences its execution at the same time as its parent task, m. Consequently, $V(m, c) = 0$. The other extreme value of $V(m, c)$ occurs when the child task c does not commence execution until the completion of the parent task m. Therefore, $V(m, c) = 1$.

- T_s: the start task set. It is assumed that each program has a unique start task.
- T_e: the exit task set. It is assumed that each program has a unique exit task.

It is assumed that all future communication between the parent and child task (after the child task is spawned) takes place synchronously with no delay in either task. This simplifies the model in the case where the communication takes place within a loop, and therefore takes place several times between the two tasks. It also simplifies the model when the communication between a pair of parent and child tasks is scattered across several locations within the parent task.

An example of how to calculate the preemption start point of a typical intertask communication is presented here. Arrange the execution of all tasks along a unified time axis, each task commences and completes at some point along this axis. The execution of two tasks may overlap due to the existence of multiple processors and the communication embedded in the middle of the (parent) task. Suppose a parent task commences its execution at time $t = 10$ and its computation time is 20 units. Therefore, the parent task completes at $t = 30$. If the child task commences execution at time $t = 16$, then the preemption start point of the child task within its parent

task is $\frac{16-10}{20} = 0.3$. That is, the child task commences execution when the parent task is 30% the way through its execution. This attribute makes sense only when data transmission does not merely occur at the end of the parent task execution. When this attribute is omitted from the task model, it implies the assumption that task execution is non-preemptive — data reception occurs at the beginning of a task while data transmission takes place at the end of a task, i.e., $V(m, c)$ for all interconnected tasks m and c is 1.

Within the preemptive task model, message-passing (transmission and reception) may occur at any location within the task. For the sake of simplicity, it is still assumed that data reception takes place before any task processing actually begins. That is, a task does not commence execution until it receives all required data; however, it can transmit data at any time before its execution completes.

The scheduling algorithm employed must choose how to deal with the preemptive task model. The preemptive task model, on one hand, can be supplied to an existing scheduling algorithm without change, thereby resulting in a distribution of tasks onto processors assuming that the tasks exhibit non-preemptive behaviour. On the other hand, a new scheduling algorithm can be developed based on the preemptive task model to take preemption into consideration. A preemptive task scheduling algorithm, PET, is presented in Section 4.4.

As stated, the preemptive task model assumes no variation between program executions. Such a model can, therefore, be precisely built by executing the program a number of times and capturing task runtime data for task attributes such as task computation time, task communication time and preemption start point. In the case where task runtime operations may take place conditionally, as discussed in Chapter 3 and [91, 136], each task interconnection requires one more attribute known as *execution*

probability. Consequently, the model becomes a *conditional and preemptive task model.* This is addressed in Chapter 6.

4.2 An Example of Preemptive Task Execution

This section compares, through an example, the system performance between preemptive task execution (PTE) and non-preemptive task execution ($NPTE$). It is assumed that, on the same processor, a task occupies computing resources until it completes its execution, i.e., no interruption of tasks takes place during execution. However, it is possible that a child task commences its execution prior to the completion of its parent task (when they are distributed on different processors).

The preemptive task model is illustrated by Figure 4.1. The parallel tasks are scheduled onto three fully-connected identical processors, as shown in Figure 4.2(a). The processor idle time during program execution is generated due to waiting for data to arrive. Such a policy is obtained by applying ERT [115] which does not consider the preemption between tasks. Runtime performance under this $NPTE$ situation is therefore $PT = 59$.

On the other hand, suppose communication (as well as task spawn) is permitted to occur at any point of the task (i.e., preemptive task execution). The preemption start point of each child task within its parent task is shown in the preemptive task model (Figure 4.1). In this PTE situation, using the same scheduling policy as that employed in the $NPTE$ situation in Figure 4.2(a), the performance is $PT = 49$, as displayed in Figure 4.2(b).

As seen, in this example, the performance is improved by 17% $((59 - 49)/59)$ by allowing task preemption to occur (i.e., in PTE situation), though under the same

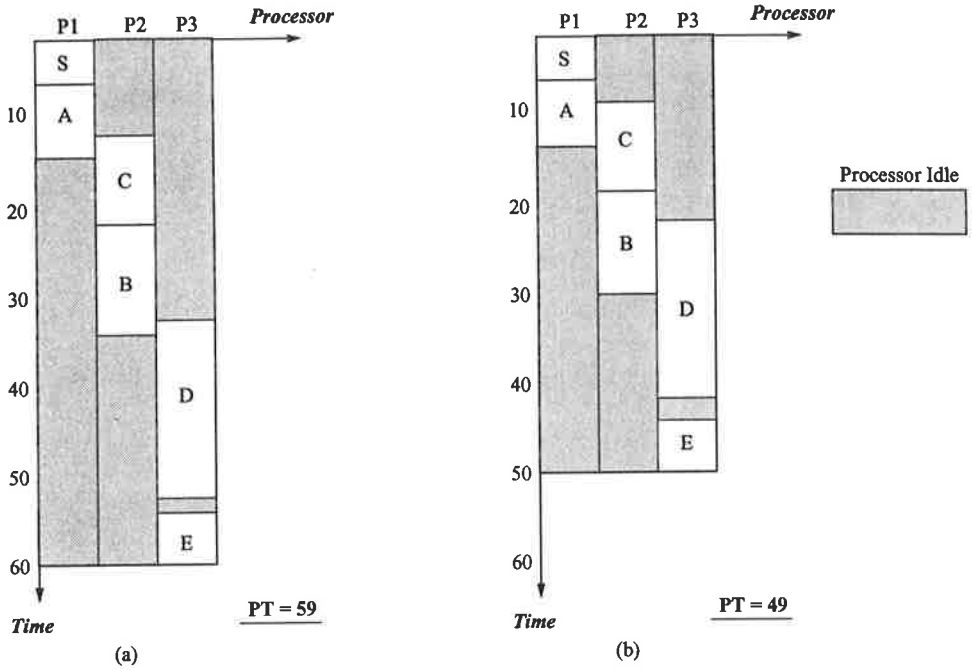


Figure 4.2. Performance of Figure 4.1 when preemption is (a) prohibited and (b) permitted during execution.

scheduling policy as that in the non-preemptive execution. This illustrates that better performance can be achieved if intertask communication is triggered as early as possible in the parent task. It remains to be seen (Section 4.4) how a customized preemptive scheduling algorithm can further improve performance.

4.3 Preemptive Task Execution

In preemptive task execution, tasks of the parallel program are distributed and arranged in execution commencement order on the underlying available processors, according to the incorporated scheduling algorithm (no matter whether it considers the preemption factor when distributing tasks). When tasks are actually submitted to execute, preemption is permitted both on the same and on different processors, depending on the strategy used for task management.

As seen, a typical feature of preemptive task execution is that the pre-determined task assignment policy is not altered while the program is executing. Only the execution sequence and commencement time of tasks (or task segments) on each host processor may vary due to variations in the time at which messages are actually transmitted.

This section investigates whether there is any performance improvement achieved through PTE, and furthermore, how to guarantee such an improvement. Section 4.3.1 illustrates two categories of preemptive task execution, and the strategies to handle the corresponding preemption. Performance gains brought about by the two strategies are elaborated in Sections 4.3.2 through 4.3.5. The efficiency discrepancy between PTE and non-preemptive task execution ($NPTE$) is studied in Section 4.3.6.

4.3.1 Two Generic Strategies For Preemptive Execution

Recall the normal definition for the "non-preemptive task execution", as adopted in [53, 151, 179, 180]: a task does not commence execution until it receives all required data; the task then executes without interruption until its completion; the task finally transmits all necessary data to its child tasks. That is to say, all data communication occurs either in the beginning or at the end of each task. Correspondingly, this chapter interprets "preemptive task execution" as encompassing the following two aspects:

- The execution of a task *may* be interrupted by another task which is distributed onto the same processor. This is termed $\alpha - preemption$ in this thesis.

- Message-passing, as well as the task spawning, operations *may* take place at any point in the task. Consequently, the commencement of a child task may be different from that in the $NPTE$ case, since it is not necessarily delayed until all its parent tasks complete execution. In this thesis, it is termed $\beta - preemption$.

Therefore, the $\alpha - preemption$ is related to the interruption in task execution, while $\beta - preemption$ brings forward or delay the execution commencement time.

It is assumed that the job scheduling algorithm of each host processor is identical. It is also assumed that an executing task can only be interrupted at the point of data transmission by another task which is ready to execute on the same processor. It is further assumed that all data reception operations occur at the beginning of the task, that is, a task does not start to execute until it receives all required data. In addition, all data transmission operations of a parent task to its child tasks are gathered together, rather than scattered throughout the whole task.

Generic strategies are proposed in this thesis to manage task execution where preemption is allowed, namely αP and βP. The αP strategy allows the occurrence of preemption on the same processor, while the $N\alpha P$ strategy does not. The concrete processing of the αP strategy can be found in the literature dealing with job scheduling [43, 70, 112], and no further discussion is presented here.

The βP strategy allows the child task can commence execution prior to the completion of its parent tasks, while the $N\beta P$ strategy does not. Briefly, βP allows the child task to commence execution once it receives all desired data and the host processor is in the idle state (i.e., no task is being executed).

Generally, the αP strategy results in α-preemption, and βP strategy results in the β-preemtion. However, both the αP strategy and the βP strategy may lead to the other form of preemption if the preemption is permitted by the corresponding strategy. The βP strategy may cause α-preemption in the case where the execution start time of a task is advanced significantly so that this task may acquire CPU resources from the task currently being executed on the same processor. Conversely, the αP strategy may cause β-preemption in a similar way: the variation in execution sequence among

tasks (segments) on the same host processor changes the actual data transmission time within a task.

4.3.2 Processor Performance $\theta(p)$

System performance PT of the parallel program is measured by the completion time of the exit task. Similarly, define the processor performance, $\theta(p)$, of a processor p as the execution completion time of the last task distributed onto p, then system performance PT is formally represented as:

$$PT = \max_{p \in P}\{\theta(p)\}$$

where P is the set of all available processors in the system. Improvement in *processor performance* on a particular processor does not necessarily incur a gain in *system performance* of the entire parallel program, which is the maximum value among all $\theta(p)$. On the other hand, the *system performance* can be improved only via enhancing *processor performance* $\theta(p)$ within the parallel system. In this thesis, preemptive task execution gains system performance by improving the processor performance θ of available processors (especially those with maximal task completion time). Therefore, this chapter investigates the influence of PTE on processor performance $\theta(p)$.

4.3.2.1 The Two Aspects of $\theta(p)$

For each processor in the parallel system, its performance θ is measured by the sum of the *task execution time* of all tasks assigned to this processor and the *processor idle time* due to delays in waiting for data to arrive. It is assumed that the *task execution time* is merely determined by the task's source code (since all processors have identical

processing speed). Therefore, once a task is distributed onto a processor, the *execution time* of this task is fixed. No effort in this thesis is made to further reduce the task's *execution time*. On the other hand, the *processor idle time* is determined by the time each task on the processor awaits incoming data (so as to commence execution). This can be reduced by an appropriate scheduling policy and execution strategy.

On each processor, two strategies, i.e., αP and βP as stated previously, can result in a change in system performance of a parallel program through altering the *idle time* of each processor. In this chapter, it is conjectured that α-preemption incurs context switching overhead between tasks (or task segments). α-preemption also causes a variation in data transmission time. On the other hand, the βP strategy is regarded as changing the commencement time of the child task, as compared to that for non-preemptive task execution. This is owing to the relaxation of the constraints on where communication operations may take place. The βP strategy may also trigger α-preemption.

4.3.2.2 Three Situations in Performance Variation

On the whole, the following situations may lead to a change in the *processor idle time*, thus *processor performance* on each processor, and subsequently *system performance*, if both αP and βP strategies apply to program execution.

- γ_1: the execution of a task, T_i, is interrupted by another task, T_j, when T_i is dealing with data transmission. The γ_1 situation reflects α-preemption taking place between T_i and T_j. In this case, the start time of T_j must be earlier than the finish time of task T_i, and there is no processor idle time between T_i and T_j.

The performance variation in this situation is denoted as $PA_s(p, T_i, T_j)$, indicating that execution of task T_i is interrupted by another task T_j on the processor p. It depends on the overhead incurred by context switching between tasks.

The performance achievement is always relative to tasks and a processor. For a certain processor p, $PA_s(p)$ denotes the total performance variation of p in the γ_1 situation. Such a convention is also followed by the remaining performance achievement notation stated below.

- γ_2: the execution start time of a task T_i is advanced due to early transmission of data from a parent task T_j, as a result of α-preemption and/or β-preemption. Correspondingly, the performance achievement of the processor is represented by $PA_e(p, T_i, T_j)$ and $PA_a(p, T_i, T_j)$ respectively.

- γ_3: the execution start time of a task T_i is postponed due to the delayed arrival of required data from parent task T_j. This is caused by β-preemption; however, α-preemption may also result in delayed data transmission. The performance variation is denoted as $PA_l(p, T_i, T_j)$, $PA_d(p, T_i, T_j)$, respectively.

From the above discussion, the performance variation caused by the α-preemption strategy on a processor is composed of three parts: PA_s, PA_e and PA_d. The β-preemption causes PA_a and PA_l performance variation.

The performance change on a processor p, denoted as $\Delta\theta(p)$, is measured as the difference in processor performance between non-preemptive execution and preemptive execution. A positive value of $\Delta\theta(p)$ indicates that preemptive execution performs better than non-preemptive execution. Similarly, a negative value indicates

performance degradation compared to the non-preemptive case. The performance gain/loss on a processor p can be calculated by:

$$\Delta\theta(p) = PA_s(p) + PA_e(p) + PA_d(p) + PA_a(p) + PA_l(p)$$

The following notation is introduced prior to a discussion on processor performance gain/loss in various situations:

- $H(p)$ is the context switch time of processor p. It arises when task execution is interrupted by another task on the same processor, and the processor needs to change context. It is assumed that $H(p)$ is identical among all processors and tasks.
- $PA(p, t_i, t_j)$ is the general performance achievement on processor p owing to preemptive task execution between task t_i and t_j.

There are three other symbols which are used in this section. They are $S_1(t,p)$, $F(t,p)$ and $U(t,p)$, the definition of which can be found in Section 3.6.1.

In Sections 4.3.3 through 4.3.5, the performance achievement caused by the situations γ_1, γ_2 and γ_3 is elaborated, when both α-preemption and β-preemption are permitted to take place during task execution. That is to say, both αP and βP strategies function at runtime. The discussion is followed by a performance comparison between different strategies in controlling preemptive task execution.

4.3.3 Performance Achievement PA_s

As aforementioned, for each processor p, $PA_s(p)$ is attained when the αP strategy is permitted to take place on p, i.e., the execution of a task can be interrupted by another

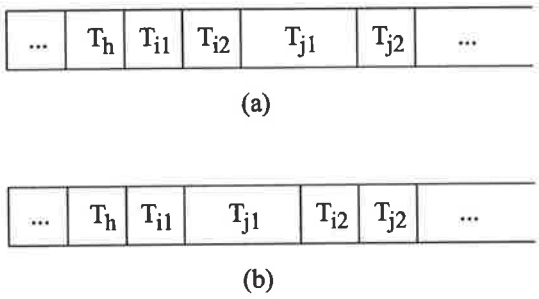


Figure 4.3. (a) Non-preemptive execution and (b) α-preemption on a processor p.

task on p. $PA_s(p)$ is measured as the difference in the completion time of the final task on processor p when comparing the preemptive execution against the non-preemptive execution.

Figure 4.3 shows the task execution on a processor p when α-preemption is not permitted (Figure 4.3(a)) and is permitted (Figure 4.3(b)) between tasks T_i and T_j on the same processor. Suppose task T_i is divided into two segments, T_{i1} and T_{i2}, by its data transmission operations, similarly for task T_j. In preemptive task execution, if the execution start time of task T_{j1} is earlier than the execution start time of T_{i2}, the first part of task T_j may acquire CPU resources from the execution of task T_i, when T_i is transmitting data.

Considering the various possible relationships between the earliest start time of task segments, T_{i1}, T_{i2}, T_{j1} and T_{j2}, there are five possible preemptive execution sequences between task T_i and T_j discussed below:

1. $(T_{i1}, T_{j1}, T_{i2}, T_{j2})$ when $S_1(T_{j1}) < S_1(T_{i2})$ and $S_1(T_{j2}) \geq S_1(T_{i2})$:

$$PA_s(p, T_i, T_j) = -2H(p)$$

i.e., two additional context switches are incurred in this task execution sequence (due to task preemption on the same processor). In addition, this execution

sequence results in early data transmission from task T_j, if the data transmission from task T_j is undertaken at the end of T_{j1}. The performance enhancement thus achieved is discussed in Section 4.3.4.

2. $(T_{j1}, T_{i1}, T_{i2}, T_{j2})$ when $S_1(T_{j1}) < S_1(T_{i1})$ and $S_1(T_{j2}) \geq S_1(T_{i2})$:

$$PA_s(p, T_i, T_j) = -H(p)$$

Furthermore, such preemptive execution causes delayed data transmission from task T_i and early transmission from T_j, the consequence of which is discussed in Sections 4.3.4 and 4.3.5.

3. $(T_{i1}, T_{j1}, T_{j2}, T_{i2})$ when $S_1(T_{j1}) < S_1(T_{i2})$ and $S_1(T_{j2}) < S_1(T_{i2})$:

$$PA_s(p, T_i, T_j) = -H(p)$$

This execution sequence also results in the early transmission of data from task T_j.

4. $(T_{j1}, T_{i1}, T_{j2}, T_{i2})$ when $S_1(T_{j1}) < S_1(T_{i1})$ and $S_1(T_{i1}) < S_1(T_{j2}) < S_1(T_{i2})$:

$$PA_s(p, T_i, T_j) = -2H(p)$$

Late data transmission from task T_i and early transmission from T_j may occur in this case.

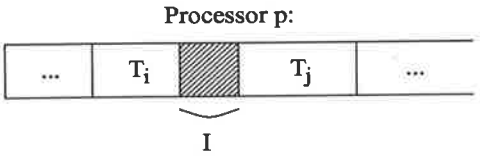


Figure 4.4. Two tasks on processor p with I processor idle time.

5. $(T_{j1}, T_{j2}, T_{i1}, T_{i2})$ when $S_1(T_{j2}) < S_1(T_{i1})$:

$$PA_s(p, T_i, T_j) = 0$$

Processor performance may vary due to the late transmission from task T_i and early transmission from T_j.

Whichever situation above occurs at runtime, the magnitude of $PA_s(p)$ is determined by the number of context switches between neighbouring tasks on the same processor. $PA_s(p)$ is always less than or equal to 0. That is to say, the processor performance θ is not improved by α-preemption in the γ_1 situation. However, in the case when the context switch time of a processor is so small that it can be ignored, $PA_s(p)$ can be regarded as approximately 0. In addition, a byproduct of the γ_1 situation is that early/late data-transmission may occur and therefore bring about further performance benefits/disadvantages as discussed in Sections 4.3.4 and 4.3.5 below.

4.3.4 Performance Achievement PA_e and PA_a

Processor performance may vary when a task T_i receives its required data earlier than predicted and permits an earlier start time than would otherwise occur in non-preemptive execution. This is the γ_2 situation as stated in Section 4.3.2. This may

be caused by either α-preemption or β-preemption. PA_2 is used to represent the sum of PA_e and PA_a: $PA_2 = PA_e + PA_a$. Figure 4.4 shows two tasks, T_i and T_j, assigned to the same processor p. I represents the processor idle time, due to task T_j's waiting for data. Denote $GR(T_j)$ as the variation in start time of task T_j gained by the early arrival of data. $GR(T_j)$ is always greater than 0, indicating an advancement in execution commencement time of task T_j. Depending on the relationship between $GR(T_j)$ and I, $PA_2(p, T_i, T_j)$ is analyzed as follows:

1. When $GR(T_j) < I$:

$$PA_2(p, T_i, T_j) = GR(T_j)$$

   There is a gain in terms of processor performance in this situation. Processor idle time still exists between task T_i and T_j, however, the idle time is less than that which occurs in non-preemptive execution.

2. When $GR(T_j) \geq I$:

$$PA_2(p, T_i, T_j) = I$$

   That is to say, in this case, the time saved due to early data arrival in task T_j is I (i.e., the idle time in non-preemptive execution). Therefore, the idle time between task T_i and T_j is completely removed. As a consequence, it may also result in further α-preemption between T_i and T_j through task interruption: the execution sequence between these task segments is determined by the start time of task segments T_{i1}, T_{i2}, T_{j1} and T_{j2} (as discussed in Section 4.3.3).

From the discussion, it can be observed that $PA_2(p)$ is positive due to the early arrival of data to task T_j in any case. That is to say, a performance gain can be guaranteed in preemptive execution situation γ_2.

4.3.5 Performance Achievement PA_d and PA_l

In situation γ_3 of preemptive task execution, the execution start time of a task may be delayed due to the late arrival of data. Both α-preemption and β-preemption can result in this situation. Actually, the message-passing delay in β-preemption is originally caused by α-preemption. Simply put the communication in the middle of a parent task will not postpone the commencement of a child task. That is to say, under the $N\alpha P$ and βP strategy, PA_l should be 0. Take PA_3 as the sum of PA_d and PA_l.

Figure 4.4 also illustrates this situation. Denote $GS(T_j)$ as the advancement in execution start time due to late data arrival. A positive value of $GS(T_j)$ indicates a gain in terms of execution time. Therefore,

$$PA_3(p, T_i, T_j) = -GS(T_j) < 0$$

Processor performance of p in situation γ_3 is non positive. This implies performance loss on processor p.

In summary, the above Sections 4.3.3 through 4.3.5 examine the variation of processor performance under three situations (γ_1, γ_2 and γ_3). Processor performance varies, depending on the sum of all these variations. These three situations are brought about by two preemptive execution strategies, i.e., αP and βP. The next section compares the performance between preemptive and non-preemptive execution so as to derive a strategy which guarantees performance enhancement in preemptive task

execution over the non-preemptive case.

4.3.6 Preemptive Execution vs. Non-Preemptive Execution

This section summarizes *processor performance* achieved through preemptive task execution, and compares such achievement against the non-preemptive task execution.

Depending on whether αP and/or βP strategy is permitted at runtime, there are, in general, different strategy combinations which control the execution of preemptive tasks, as shown below:

1. $N\alpha P^* N\beta P$: neither αP nor βP strategy is applied at runtime. Data reception takes place at the start of the task, and data transmission at the end. Furthermore, the execution of the task is not interrupted. This is equivalent to non-preemptive task execution and has been adequately addressed in other work [128].

2. $N\alpha P^* \beta P$: a task can start before the completion of its parent tasks due to internally-positioned data communication operations (i.e., communication is not restricted to the start or end of the task), but it is not permitted to interrupt the execution of other tasks on the same processor. That is to say, once a task gains access to the CPU resources, it executes until completion.

3. $\alpha P^* \beta P$: the execution of a task can be interrupted by another task on the same host processor, due to α-preemption, β-preemption or both. Data transmission is permitted in the middle of task execution.

The combination of strategies αP and $N\beta P$ is not practical. On one hand, the study of preemptive task execution assumes that α-preemption occurs at the point of data

$PA(p)$	$N\alpha P^*N\beta P$	$\alpha P^*\beta P$	$N\alpha P^*\beta P$
PA_s	0	< 0	0
PA_e	0	> 0	0
PA_d	0	< 0	0
PA_a	0	> 0	> 0
PA_l	0	< 0	0
$\Delta\theta(p)$	0	?0	> 0

Table 4.1. Performance comparison between non-preemptive and preemptive executions.

transmission. It is only meaningful if the data transmission is placed in the middle of the task, where interruption may occur. On the other hand, the $N\beta P$ strategy restricts data communication to the two ends of the task (reception in the beginning and transmission at the end). Consequently, $\alpha P^*N\beta P$ is excluded from the strategies employed in handling preemptive task execution.

Table 4.1 compares *processor performance* of an available processor across different preemptive strategies against non-preemptive task execution. Recall that $\Delta\theta(p)$ is the sum of all performance variations. According to the definition, *system performance* of the parallel program, PT, is the maximum value among all these *processor performances*, $\theta(p)$.

From Table 4.1, it is observed that, for any available processor p in the parallel system, processor performance $\theta(p)$ in $N\alpha P^*\beta P$ preemptive execution is guaranteed to be better than that in non-preemptive execution. On the other hand, $\alpha P^*\beta P$ does not always provide superior performance to $N\alpha P^*N\beta P$, because of possible delays in data-transmission due to α-preemption, thus a question mark is placed in the corresponding $\Delta\theta(p)$ row. In summary, by merely applying βP to parallel task execution, it is expected that system performance is improved, in comparison to non-preemptive execution.

4.4 Preemptive Task Scheduling

In addition to preemptive task execution, another aspect studied in this chaper is the preemptive task scheduling problem. This is realized through the construction of the preemptive task model and a preemptive task scheduling algorithm.

Preemptive task scheduling stresses that the scheduling algorithm should consider the existence of preemption among task execution; this has been largely ignored in previous work. The task model in preemptive task scheduling is illustrated with one additional attribute, named *preemption start point*, as well as the standard attributes of task computation time and communication time between tasks as addressed in many other algorithms such as [53, 115, 128, 151]. Such a task model more accurately reflects the execution of parallel tasks than those currently used. The construction of the preemptive task model, if required, for a parallel program follows the same strategy as that discussed in Section 3.5. In particular, the estimation of the attribute *preemption start point* adopts the linear regression model, similar to the task computation time and inter-task communication time attributes.

A new preemptive scheduling algorithm, *PET*, is proposed to deal with the preemptive task model. At any instant, each *schedulable* task or *free* task (i.e., those tasks for which all parent tasks have been distributed onto processors) in the parallel program is assigned a priority value based on its "earliest start time". The *PET* algorithm is outlined as follows. Notation is found in Section 3.6.1.

1. Initialization:

   $\forall t_i \in T, Q(t_i) =$ the number of parent tasks of t_i

   $\forall p_j \in P, A(p_j) = 0$

   $\forall t_i \in T, G(t_i) = (Null, Null)$

Set current time $t = 0$.

2. Task Scheduling:

   - Let W be the set of all tasks that have not yet been scheduled and whose parent tasks have all been scheduled. That is to say,

     $$W = \{t \mid t \in T, Q(t) = 0, G(t) = (Null, Null)\}$$

   - If $\| W \| = 0$, then exit from the scheduling process.

   - Select a task, t_k, in W and a processor, p_l in P, so that the task t_k on p_l has the smallest value of the "earliest start time" which can be calculated by:

     $S_2(t_i, p_j) = \max\{S_1(t_i, p_j), A(p_j)\}$, where

     $S_1(t_i, p_j) = \max\{(F(t_{i1}, p_{j1}) - U(t_{i1}) \times (1 - V(t_{i1}, t_i)) + M(t_{i1}, t_i))\}$, for every t_i's parent task, t_{i1}, on its host p_{j1}.

     Evaluate the smallest "earliest start time" of task t_k on processor p_l, among all tasks ready to be scheduled on all available processors, by:

     $$S_0(t_k, p_l) = \min\{S_2(t_i, p_j) \mid \forall t_i \in W_e, \forall p_j \in P\}$$

     Task t_k is then assigned onto processor p_l.

3. Update the following variables:

   $N(p_l) = N(p_l) + 1$

   $G(t_k) = (p_l, N(p_l))$

   $A(p_l) = A(p_l) + U(t_k, p_l)$

$W = W - \{t_k\}$

For any child task t_i of task t_k, $Q(t_i) = Q(t_i) - 1$. If $Q(t_i) = 0$, then $W = W \cup \{t_i\}$.

4. Go to Step 2.

The complexity of the PET algorithm is $O(mn^2)$ where m is the number of processors in the parallel and distributed system and n is the number of tasks of the parallel program. The derivation procedure is the same as that undertaken in Section 3.6.2.

The experimental results regarding preemptive task scheduling, as well as preemptive task execution, can be found in Section 7.3. Generally, the preemptive task scheduling strategy, cooperating with the preemptive task execution strategy, can achieve better system performance than the non-preemptive scheduling and task execution management policy. That is to say, prior to program execution, the PET algorithm schedules parallel tasks, with consideration for task preemption. Furthermore, at runtime, the job scheduling policy of each available processor adopts the $N\alpha P^*\beta P$ strategy to handle the task execution so that, on the same processor, a running task executes to its completion, without being interrupted by other tasks.

Chapter 5

Conditional Parallel Programming Support

A large number of parallel algorithms, computational models and machine architectures have been proposed to realize parallel processing [163]. It is highly desirable to translate such algorithms or theoretical research into operational programs on physical parallel and distributed systems — through parallel programming.

This chapter presents work undertaken for the support of parallel program development. In particular, this chapter focuses on the support for *conditional parallel programming*; that is to say, the runtime operations between parallel tasks of an application program may be associated with conditional branches. Therefore, the behaviour of parallel tasks may not be determined until runtime. Static precedence and communication relationships may not be fully exercised in particular program execution.

Support for parallel program development has been studied for some time and a number of environments or tools have been proposed. These include HeNCE [15], Code [22], PTOOL [4], Linda [2, 34], CORBA [129, 130], MPI [121] and P4 [27, 28]. A

detailed review is given in Chapter 2. Application programmers are, to some extent, relieved of various aspects of tedious and complex parallel program development. Nevertheless, support for *conditional parallel programming* has been largely ignored. It is basically the application programmer who must deal with the new challenges arising from conditional parallel programming.

The programming support discussed in this thesis is provided through a library of primitives, referred to as *ATME primitives* (or the *ATME library*). The *ATME library* is a set of routines which are embedded in user tasks to utilize the service provided by the *ATME* environment. *ATME* utilizes *PVM* (Parallel Virtual Machine) as a basis, and more significantly, extends *PVM* so as to support conditional parallel program development and automatic task scheduling. Through the availability of the *ATME* primitives, the application programmer is able to concentrate on the design of *algorithms* to solve the problem, without having to address subtle *operational* issues (i.e. implementation details) in conditional parallel program implementation.

The parallel program is represented by a task model, as illustrated in Section 3.2, composed of a number of interrelated tasks. Each task of the program realizes certain behaviour with respect to the application, and communicates with other tasks for required data. This thesis assumes that each task is an atomic execution unit. Furthermore, the communication between interrelated parallel tasks is presumed to be implemented via message-passing operations.

This chapter is arranged as follows. Section 5.1 briefly describes the major points of *PVM*, which is widely adopted to realize parallel processing and acts as the basis for advanced programming support (such as *ATME*). New issues in conditional parallel programming are discussed in detail in Section 5.2. The *ATME* primitives and related issues are presented in Section 5.3. Section 5.4 illustrates the core of the *ATME*

primitives implementation, i.e., the execution monitor. Section 5.5 discusses the work flow of three typical *ATME* primitives: conditional task spawn,. conditional data transmission and conditional data reception, with a focus on their communication with the *execution monitor.*

5.1 Parallel Virtual Machine (*PVM*)

This section briefly summarizes the Parallel Virtual Machine (*PVM*), a software package for assisting in the development of parallel programs in a (heterogeneous) distributed computing environment. *PVM* is the fundamental infrastructure on which *ATME* realizes its support for conditional parallel programming and the automation of the conditional task scheduling process.

Section 5.1.1 illustrates the *PVM* computation and system models, as well as the mechanisms facilitated within *PVM*. Section 5.1.2 presents the *PVM* user interface, i.e., a runtime library which can be embedded within application programs.

5.1.1 *PVM* Models

PVM emulates a generalized distributed memory multiprocessor in (heterogeneous) networked environments. *PVM* presents the user with a virtual parallel machine, through which the programmer can regard all processors in the system as identical and fully connected. The user is therefore able to concentrate on the design of parallel programs and algorithms while not paying excessive attention to system architecture details.

From *PVM*'s perspective, an application program, written in either C or Fortran, is composed of a number of partitioned tasks, among which exist precedence relationships,

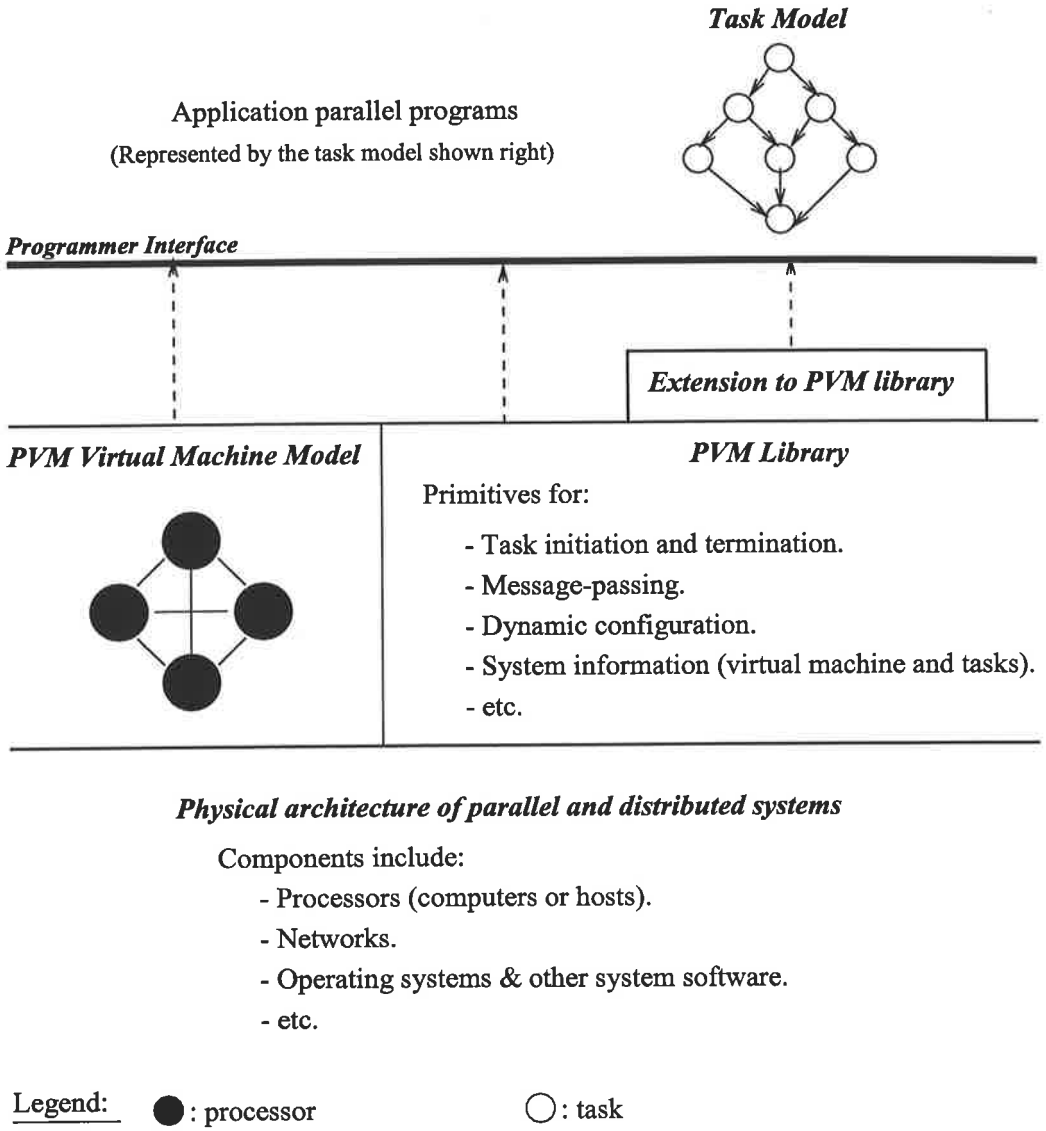


Figure 5.1. Layers of software in supporting parallel programming in PVM.

and which communicate with each other via message-passing. A PVM task is assigned to a host (processor). Each task is an independent execution unit.

Figure 5.1 shows the computation model and the architecture view of PVM. As seen, PVM makes the underlying system transparent to the user and simply presents a *virtual machine model.* Functionally, PVM contains two parts: the *daemon* and the *library* (discussed in Section 5.1.2). The user is required to provide a *host file* which lists all participating hosts in the virtual machine. PVM sets up a *daemon* process on

each host included in the host file in order to construct a parallel computing system and manage the communication between hosts. The *PVM* application program can be started from a shell command line on any of registered hosts.

5.1.2 *PVM* Library

PVM provides a library of primitives for the programmer's disposal. Primitives enable a user task to dynamically add and delete hosts from the virtual machine, spawn and terminate other tasks, synchronize with and send messages to other tasks allocated on the same or different hosts. In addition, *PVM* provides facilities to gather a number of cooperating tasks together into a "task group". Primitives exist which allow a task to dynamically register with and leave a task group. Furthermore, the programmer can obtain information about the virtual machine configuration and active *PVM* tasks. The detailed *PVM* routines are found in Appendix B of the user manual of *PVM* [69].

With regard to distributing spawned tasks onto the underlying available hosts, *PVM* (version 3.3) employs a round-robin algorithm (more discussion is undertaken in Section 5.2). Actually, *PVM* (version 3.3) does not specifically strive for an efficient task scheduling policy through which high performance is achieved from the underlying parallel and distributed system. It allows for further extension.

The conditional programming support provided in this thesis is based on *PVM* version 3.3, which is the latest version when the project commenced in 1995. In the following sections, new issues arising from conditional parallel programming are examined. The *ATME* library, which is established on top of *PVM* structure, and technical details are also presented.

5.2 New Challenges

In conditional parallel programming, when conditional branches can be associated with task runtime operations, application programmers face more challenges than ever. From the user's perspective, these new issues have no direct and efficient solution in support tools available to date.

This section highlights the new challenges involved in conditional parallel program development, with a focus on the processing required to enable conditional task scheduling automation, conditional task spawn, conditional data transmission and conditional data reception. Approaches to solve these problems are stated in Section 5.3.

5.2.1 Task Scheduling Automation

Task scheduling is one of the most complicated and tedious issues in parallel programming. According to El-Rewini *et. al.* [52], the process of task scheduling should be automated in order to free the application programmers from this tedious and unnecessary complexity, while still attaining high system performance. In this thesis, this objective is achieved mainly through automating the task scheduling algorithm to efficiently distribute parallel tasks onto the underlying available processors.

In version 3.3, PVM adopts a *round-robin* algorithm to schedule parallel tasks onto the underlying machine. This algorithm places a free processor in a $FIFO$ (First In First Out) queue once it completes the execution of its assigned task. When a task is spawned and a processor is required, PVM selects the head of the queue and assigns the task to it. The *round-robin* algorithm has its advantages: it is simple and little computation is involved in its implementation. In addition, such

an algorithm makes dynamic task distribution possible, without incurring excessive runtime overhead. However, this scheduling algorithm does not consider the special requirement of tasks and processors, and therefore generally does not achieve high system performance (as manifested in the experiment performed in Section 7.2.3). For example, two communication-intensive tasks (say, successively spawned) are prone to be allocated to different hosts according to the round-robin algorithm, and therefore incur extra communication overhead at runtime.

This thesis focuses on static task scheduling and believes that task scheduling is an integrated process which requires the cooperation of a number of factors. The study of the scheduling algorithm itself is not enough. A task model and a processor model, which adequately reflects the actual requirements of the parallel program and distributed system, are basic factors demanded by the scheduling algorithm in order to produce an efficient scheduling policy.

In the case of conditional parallel programming, the task model of the program in different executions is not identical, due to conditional branches attached to task runtime operations. As a consequence, the task model is not precisely known prior to execution. This brings more challenges to the automation of the task scheduling process. In this thesis, Chapter 3 studies the scheduling problem in the presence of conditional branches and provides a practical approach to deal with this problem. The estimation of the task model of the conditional parallel program can expect high accuracy when the task model, particularly the task attributes, does not vary significantly between program executions. Experimental results, presented in Chapter 7, reveal that the proposed strategy can generally achieve much higher system performance than a random distribution strategy and the round-robin algorithm, both of which do not consider the subtle requirements of tasks and processors.

This chapter puts into practice the theoretical research conducted on the conditional task scheduling problem (in Chapter 3) through an $ATME$ runtime primitive, i.e., *tme_spawn* discussed in Section 5.3.3.

5.2.2 Conditional Task Spawn

Conditional task spawn refers to a spawn operation which is associated with conditional branches. As a result, it can not be determined, prior to execution, whether such a task spawn actually takes place at runtime or not. A task may not necessarily run in *every* program execution.

With respect to the coding of a conditional parallel program, it should be noted that even an unconditional spawn of a task is not guaranteed to result in a new task. This is because the parent or spawning task may itself be conditionally spawned. Figure 5.2 illustrates this situation through three tasks: task C and its two parent tasks A and B. Since tasks A and B may also be conditionally initiated by their parent tasks, it is not known until runtime which task (A or B) actually executes. Therefore, it is necessary that both A and B be responsible for spawning task C. However, at runtime, only one of either task A or B can actually spawn task C, or else two instances of task C will be generated; this does not meet the requirements of the application described in Figure 5.2.

New issues are consequently raised in conditional task spawn as mentioned above. Suppose in the example shown in Figure 5.2, task A attempts to spawn C. In order for the program to execute properly, it should detect whether or not task C has already been spawned by C's other parent tasks (i.e., task B). However, PVM has no mechanism to determine whether a task (say C) has been spawned or not. Furthermore, there is no primitive available which can provide information regarding

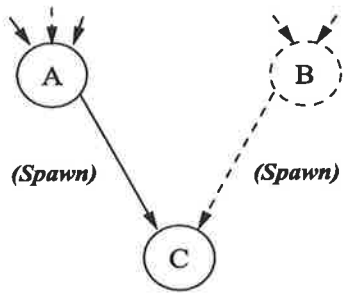


Figure 5.2. Conditional task spawn.

whether a task (say C) is *being spawned* by its parent task(s). Hence, it is the programmer's responsibility to capture and keep such information for later reference while the program executes. The detection and management of such task information is not only complicated and tedious, but also presents an additional challenge to the application programmer. Moreover, the programmer should guarantee that task C is uniquely spawned by either A or B. In the case of large parallel applications, this additional work is considerable.

Fortunately, all the above issues in conditional task spawn are related to the *implementation* of the parallel program. Such issues are encountered by all conditional parallel programs, and can be automated.

5.2.3 Conditional Data Transmission

Conditional data transmission refers to a data transmission operation associated with conditional branches. Conditional data transmission may also be involved with task spawn operations, as explained through an example below.

Figure 5.3 portrays conditional data transmission between task A and C. C has another parent task B which may or may not spawn C at runtime. Before actually sending data to task C, task A must detect whether C has been spawned or not. If not, task A needs to first spawn task C and then perform data transmission. If, on the other

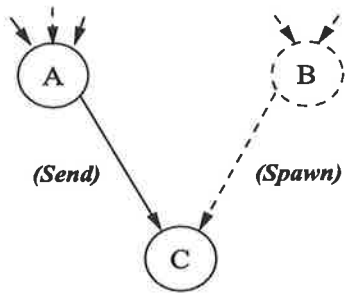


Figure 5.3. Conditional data transmission.

hand, task C has already been spawned by another parent task, B, task A still needs to obtain the unique identification of task C so as to be able to transmit the data, since task identification is required by PVM's point-to-point data communication. Such processing has no direct support in the PVM runtime library when task C is spawned by tasks other than A. As a consequence, additional communication (on top of what is required by the application) between user tasks is needed to obtain such information, before task A can actually pass messages to its child task C.

5.2.4 Conditional Data Reception

Owing to conditional branches associated with the data transmission operations, data reception may also be conditional. In the case of synchronized data reception, a child task is suspended to wait for the arrival of the required data from its parent task.

Figure 5.4 portrays conditional data reception. When task C (suppose it has other parent tasks as well) tries to accept data from its parent task B, C should know beforehand whether B has been spawned or not, as well as whether or not B has transmitted, or will send, data to it. With such information, task C can then determine whether to skip such a data-reception operation and its related operations. Such evaluation relies on the runtime task state information, which, similar to conditional task spawn and data transmission, is mostly related to operational issues, rather than

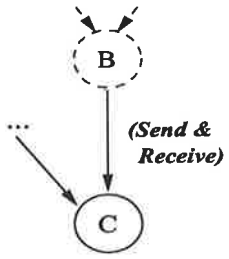


Figure 5.4. Conditional data reception.

application specific requirements. In addition, there is no support with the PVM tool for this problem.

From the discussion in Section 5.2, it can be seen that the new issues raised in conditional parallel programming stem from permitting conditional branches to be associated with task runtime operations. Furthermore, all these issues are generally related to program *implementation* (operational aspects) and not generally concerned with the *design* (functional aspects) of algorithms and programs. An environment, or a support tool, **can** provide assistance so that the programmer is freed from considering these cumbersome issues.

There are a variety of approaches which may be adopted to provide support for conditional parallel programming. There are basically three types of strategy, as examined in Section 2.4.1. Briefly, they are compiler modification, a graphical and user-friendly environment, and a runtime library. This thesis provides the user with a set of runtime primitives, considering its virtues of being easy and flexible to use. An environment is also presented (Chapter 6) to provide complete support. Primitives can be embedded into tasks in a straightforward way, and thus significantly reduce the work load on the programmer.

5.3 *ATME* Library

This section discusses the *ATME* library which is designed to tackle the challenges stated in Section 5.2. The contribution of the *ATME* library is presented through three major primitives, namely, task spawn, data transmission and data reception, respectively. These three *ATME* primitives are similar in form to their counterparts within the *PVM* runtime library, but incorporate more complex functionality.

Section 5.3.1 states the enhancement of *ATME* primitives over those of *PVM*. Section 5.3.2 illustrates the information required from the programmer in order to use *ATME* primitives. The *ATME* library is elaborated in Section 5.3.3.

5.3.1 Extensions to *PVM*

It is well known that *PVM* has achieved wide acceptance. *PVM* has been adopted in a number of projects, such as groundwater research and atmospheric modeling [164]. One important motivation for the use of *PVM*, as summarized by *PVM* developers, is the cost-effective performance of cluster computing systems [163]. It has been revealed that "generally, clusters are about 10 times as cost-effective as supercomputers for a given performance capability, for several classes of applications" [164]. Other reasons for the increasing use of *PVM* rest in its high degree of portability and its straightforward and robust user interface (primitives) which meet application requirements. *PVM* has incorporated more and more machine architectures in its recent releases. Consequently, *PVM* can be employed to construct a transparent virtual machine, based on a large variety of architectures. So far, *PVM* can run on, to list a few, CM-5 (introduced in *PVM* version 3.2), UXPM (Fujitsu M780 UXP/M, version 3.3.4), Pentium (version 3.3.6), IBM SP-2 (version 3.3.8) and WIN32 (version 3.4.0) [110]. A detailed list of

PVM supported architectures and operating systems is found in [111].

The *ATME* library enriches *PVM* functionality in two ways. One is the extension of the *PVM task spawn* primitive to tackle the issue of conditional task spawn. The *ATME* library cooperates with other *ATME* functional components (elaborated in Chapter 6) to realize task scheduling automation. The *ATME* environment adopts an efficient scheduling algorithm which produces a scheduling policy prior to program execution, based on the estimated task model of the program. Currently, *ATME* utilizes the algorithm *CET* (Section 3.6.2), which deals with the conditional task scheduling problem and its efficiency is verified through extensive simulated experiments (Chapter 7). The *ATME*-generated policy file can be dynamically accessed by *ATME* primitives inside user tasks. As a result, *ATME* can read the scheduling policy file to allocate newly-spawned tasks, and thus realizes automated conditional task scheduling of a parallel program.

The other extension of *ATME* over *PVM* relates to *data transmission* and *data reception* primitives. *ATME* provides more functionality than *PVM* does, in order to deal with conditional message-passing between tasks. At present, *ATME* simply deals with point-to-point communication, and leaves the study of the multicasting and broadcasting communication to future work which can make use of the mechanisms presented in this thesis.

The contribution of *ATME* stated above is reflected in three major primitives: spawn, point-to-point data transmission and reception, denoted as *tme_spawn()*, *tme_send()* and *tme_recv()* respectively.

Available Processor

rabbit
magpies
woozle

Figure 5.5. The host file.

5.3.2 User Input

This section identifies the information required from the user who utilizes the *ATME* services. This section distinguishes between the *ATME primitives* and the *ATME environment* which in most cases can be distinguished from each other through context. The former refers to a set of the runtime subroutines (functions) which are incorporated in the program source code, while the latter refers to the whole environment presented to a programmer who utilizes the *ATME* primitives to deal with, say, parallel task communication. User information *directly* employed by the independent *ATME* primitives (i.e., those used in isolation in most of the components of the *ATME* environment) and the *ATME* primitives embedded in the *ATME* environment may not be identical.

The application *program*, which utilizes either the *ATME* environment or *ATME* primitives, is a mandatory input. The program is partitioned into tasks, each of which is a stand-alone sub-program and incorporates intertask communication. The program is written in C, as supported by *ATME* and the underlying *PVM* architecture. The application program is represented by a task model, as discussed in Section 3.2.

In addition to the program itself, a *host file* which lists all available processors comprising the parallel and distributed system (virtual machine, from the user's perspective) is required by the *ATME* environment. An example of the host file is shown in Figure 5.5. The name of processors or machines (as recognized by the

Parallel Task	*Scheduled Processor*	*Execution Sequence*
{task S}	{rabbit}	1
{task A}	{rabbit}	2
{task C}	{magpies}	1
{task B}	{magpies}	2
{task D}	{woozle}	1
{task E}	{woozle}	2

Figure 5.6. The scheduling policy file.

system and the network) is presented in the host file. When starting PVM, *pvmd* (PVM daemon) is generated on each processor, as listed in the host file, establishing the virtual machine as well as managing inter-processor communication.

Depending on whether the user program employs $ATME$ primitives separately or not, there are two types of target machine files which are *directly* utilized by $ATME$ primitives. If, on one hand, only the $ATME$ primitives are used in the user program, the host file directly accessed by the $ATME$ primitives is identical to that provided by the user (i.e., as shown in Figure 5.5). In this situation, issues in conditional parallel programming (as presented in Section 5.2) are largely addressed by the $ATME$ primitives. However, the programmer must be satisfied with the default round-robin scheduling algorithm supplied by PVM.

If, on the other hand, the $ATME$ primitives are utilized in conjunction with the other functional components of the $ATME$ environment, then the programmer provides the *host file* (Figure 5.5) to the entire $ATME$ environment. Such a host file is transparently converted into a *scheduling policy file* by the $ATME$ environment, as shown in Figure 5.6, which is then directly accessed by the $ATME$ primitives. The *scheduling policy file* lists the tasks of the program, available processors and the allocation strategy of the tasks onto processors. The execution sequence of tasks

assigned on the same processor is also illustrated. With this *scheduling policy file*, the programmer is offered automated task distribution by the scheduling algorithm through the *ATME* environment, on top of the support in conditional spawn and communication (provided by the *ATME* primitives).

5.3.3 *ATME* Primitives

ATME provides the programmer with a set of primitives to handle task runtime operations in conditional parallel programs. Such operations include task spawn and exit, data transmission and reception, and information requests regarding active tasks and system configuration.

Prior to a formal discussion of the *ATME* primitives, the unique task identification mechanism employed within *ATME* merits some explanation. This is needed by both application problems and the underlying *PVM* platform. On one hand, user-provided parallel tasks must be uniquely identified within *ATME*, in order to realize point-to-point communication. *ATME*, as the programmer interface, manages all tasks within a table. Each task is uniquely identified by the *task index number* (*tidxno*) within the table. On the other hand, *PVM*, the environment on which the user tasks are to be physically executed, dynamically assigns each spawned task a *task identification number* (*tidno*) and utilizes it throughout program execution. In *ATME*, the conversion from *task index number* into *task identification number* is completely automatic and hidden from the user.

Table 5.1 shows a few typical *ATME* primitives with respect to task spawn and message-passing. The initiation of a task is realized by the primitive *tme_spawn*. A user task, at this point, is marked with its *task name*. *ATME* returns a unique task index number *tidxno* (later translated into the *tidno* used by *PVM*) to the user, which

ATME runtime primitive	Brief Explanation
tme_spawn(char *tname, char **argv)	Spawn a task with the name as *tname*; $ATME$ returns an integer as this task's index number, tidxno (used within $ATME$).
tme_exit()	The task leaves $ATME$, but still exists as a process in the system.
tme_initsend(int encoding)	Prepare to send data with encoding scheme identified by *encoding*.
tme_send(int tidxno, int msgtype)	Send data to the task identified by *tidxno*. The message is distinguished with *msgtype*.
tme_recv(int tidxno, int msgtype)	Receive data from the task *tidxno*. The message is marked by *msgtype*.
tme_pk$T$(T *dp, int nitem, int stride)	Pack an array of the given data type T into the message sending buffer. Different types of data can be packed in the same message buffer. T can be int, float and byte etc.
tme_upk$T$(T *dp, int nitem, int stride)	Unpack data in message receiving buffer into the corresponding data array (according to type T.)

Table 5.1. ATME runtime primitives.

is utilized as the unique identification of the user-spawned task.

The same source code can have several instances within $ATME$ at runtime, which are distinguished by the *tidxno*. The phenomenon of multiple task instances is frequently encountered in applications such as scientific computation. Matrix multiplication is a typical example. In such computation, each of the two matrices is partitioned into several sub-matrices. Multiplication is first conducted between sub-matrices, from the original matrix respectively, the results of which are used to do further computation, if required. The multiplication of each pair of sub-matrices can be done independently, since synchronization of data can be largely avoided by proper partitioning of the matrix. This indicates that parallelism can be achieved in such a calculation. Furthermore, the multiplication of sub-matrices can be realized through the same task (say, named $MULT$), by providing it with different sub-matrices. Consequently, a number of $MULT$ tasks are spawned in the system, each of which

```
/*-------------------------------------------------------------*
         User code to call tme_spawn: task A spawns task B.
 *-------------------------------------------------------------*/
  // cond1() simulates conditions associated with the task spawn
  // operation.
  if  (cond1() == 1)    {

    /* Spawn task B, no need to worry about duplicated task
     * initiation, since ATME does the transparent check.
     * ATME returns an integer value as the unique index number
     * of this task.
     */
    numt = tme_spawn("B", para);

    if  (numt == 0)   {
      error_handling("error in spawn...");
    }

      ...

  }
```

Figure 5.7. Code of conditional task spawn.

executes in parallel to the others.

During the spawn of a parallel task, parameters can be transferred to the child task. *ATME* automatically selects a processor on which this task is to be distributed, either from the *scheduling policy file* or from the *host file* (discussed in Section 5.3.2), depending on whether or not the *ATME* primitives are employed along with other functional components of *ATME* environment. An example code fragment illustrating conditional task spawn is given in Figure 5.7. The corresponding task model is shown in Figure 5.2. As seen, there is little to be done by the programmer. It is *ATME* (not the programmer) that guarantees a task is uniquely spawned by only *one* of its parent tasks. Related technical details are discussed later in Section 5.5.1.

The termination of a spawned task is undertaken via the primitive *tme_exit*. This primitive tells the local deamon process (*pvmd*) that this task is exiting $ATME$ as well as PVM. However, this task still exists in the system, as a process, within PVM.

Prior to data transmission, the task invokes the primitive *tme_initsend* to clear the message buffer attached to the task and to specify the encoding scheme of the communicated messages. This encoding scheme must be known to both the sending and receiving tasks, in order to properly recognize the data transmitted.

Primitives *tme_send* and *tme_recv* realize point-to-point asynchronous blocking transmission and reception. In $ATME$, data is packed into the *sending buffer* before it is sent off (also known as "marshalling") and is unpacked after it is accepted by the *receiving buffer* (also known as "unmarshalling"). The *asynchronous blocking data transmission* refers to the fact that the sending operation returns as long as the sending message buffer is free, irrespective of whether the receiving task is ready or not. On the other hand, the *asynchronous blocking data reception* returns when the data is in the receiving message buffer, regardless of the state of the sending task. In this way, the communication can take place concurrently with computation. Data of different types require different pack/unpack primitives, hence $ATME$ provides primitives *tme_pkT* and *tme_upkT*, where T represents data types which are permitted in C. Within a transmitted message, it can mix data of different types. A message from a task is identified by the *message type* (an integer value) in order to distinguish it from other messages sent/received by the same task.

Examples of conditional data transmission and conditional data reception are shown in Figures 5.8 and 5.9, respectively. As seen, the programmer can simply send a series of data values in the same message, unconcerned about the state of the receiving task, since $ATME$ automatically detects the receiving task and spawns it if the target task

```
/*-------------------------------------------------------------*
          User code to call tme_send: task A sends data to B.
 *-------------------------------------------------------------*/
  /* Get the task index number of the target (receiving) task;
   * getchdtidxno() is an ATME-specific function.
   */
  chdtidxno = getchdtidxno(mytidxno, "B", -1);

  // cond2() simulates conditions associated with the following data
  // transmission operation.
  if  (cond2() == 1)   {
    /* Pack the to-be-transmitted data into the message buffer.
     * Note that different types of data can be packed into the
     * same message buffer.
     */
    strcpy(f_str, "A to B");
    tme_pkstr(f_str, 1, 1);
    tme_pkint(f_intp, 1, 1);
    tme_pkdbl(f_dbp, 1, 1);

    /* The programmer is freed from considering whether or not
     * task B exists, since ATME implements such checks and
     * issues the task-spawn operation if B has not been spawned
     * yet.
     */
    f_ret = tme_send(chdtidxno, MSGTYPE2);

    if  (f_ret == -1)   {
      error_handling("error in sending...");
    }
  }

    ...
```

Figure 5.8. Code of conditional data transmission.

```
/*-------------------------------------------------------------*
   User code to call tme_recv: task B receives data from task A.
 *-------------------------------------------------------------*/
  /* Get the task index number of the sending task;
   * getpartidxno() is an ATME-specific function.
   */
  partidxno = getpartidxno(mytidxno, mytidno, partidxno);

  /* ATME skips the following receiving operation and related
   * statements afterwards, if it detects that nothing is transmitted
   * from its parent task A.
   */
  f_ret = tme_recv(partidxno, MSGTYPE2);

  if   (f_ret == -1)  {
    error_handling("error in receiving...");
  }

  // Unpack received message into original data type
  tme_upkstr(f_str, 1, 1);
  tme_upkint(f_intp2, 1, 1);
  tme_upkdbl(f_dbp2, 1, 1);

  ...
```

Figure 5.9. Code of conditional data reception.

has not been started. In data reception, $ATME$ automatically skips the reception as well as related operations, if it detects that the data is not going to be transmitted by its parent task. Technical details are elaborated in Section 5.5.2 and 5.5.3 respectively.

With respect to other primitives such as the dynamic addition and deletion of hosts and the obtaining of the active task and system information, $ATME$ employs similar forms to those in PVM. These can be found in Appendix B of [69].

5.4 Execution Monitor

A core mechanism which realizes conditional parallel programming support is the design of an *ATME* self-contained task, named the *execution monitor*. This task runs concurrently with user-provided tasks and supplies runtime information to enable *ATME* to solve the difficulties raised in conditional programming. This section addresses the design and implementation issues in the *execution monitor* task.

Section 5.4.1 illustrates the translation of a user-provided parallel program (represented by the task model) into *ATME* tasks. Messages transmitted between the *execution monitor* and user tasks, are called "events" in this thesis, and are introduced in Section 5.4.2. Section 5.4.3 presents design and implementation details of the execution monitor.

5.4.1 User Task Preprocessing

The user-provided application program is preprocessed before it is actually executed on the underlying target machine. Preprocessing of user tasks is conducted transparently to the user by the *ATME* environment (discussed in Chapter 6). *ATME* wraps each user-supplied task in code which deals with the conditional communication to other tasks on the *PVM* platform. In addition, *ATME* introduces additional tasks and establishes parameters, which are used by *ATME* itself in order to implement the programming support.

Program analysis, which focuses on the instrumentation of the program, is conducted on the parallel tasks. The aim is to capture task runtime information so that it can be used in generating a scheduling policy for the conditional parallel program. This is discussed further in Chapter 6.

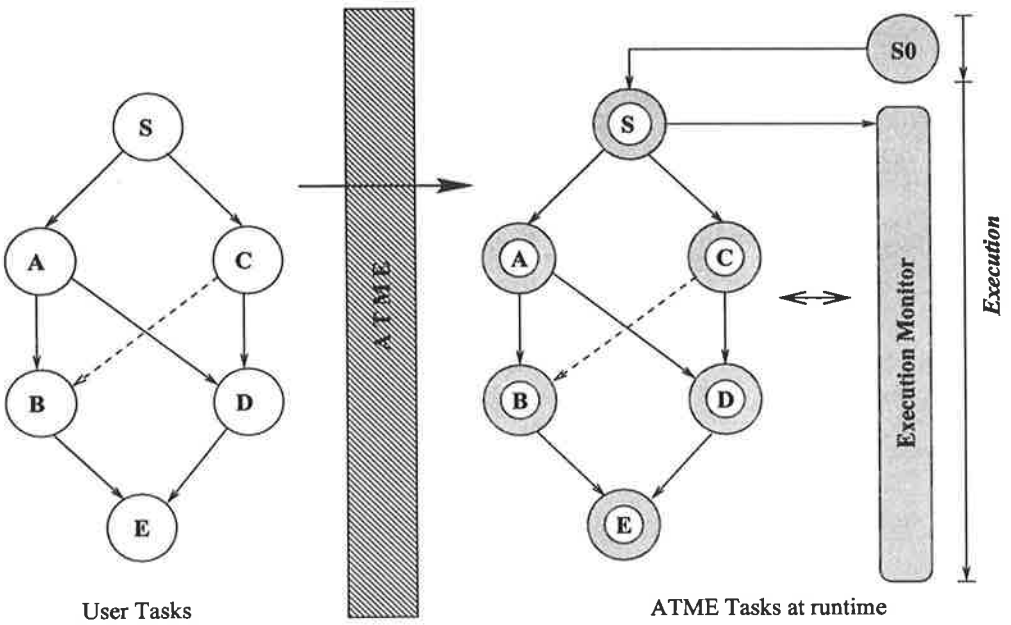


Figure 5.10. Preprocessing of user tasks into $ATME$ tasks.

Figure 5.10 graphically illustrates the transformation of the user-supplied tasks into $ATME$ tasks. The shaded parts in the figure represent $ATME$ extensions to the user tasks. $ATME$ changes neither the precedence relationships between tasks defined by the application programmer, nor the source code with respect to its functionality. $ATME$ merely extends the user code to handle conditional task spawn and communication.

$ATME$ introduces two tasks for its own purposes. One task is the initial task, S_0, which starts the user program by distributing the *start task* in the program, i.e., S in Figure 5.10, onto a predefined processor (according to either the host file or the scheduling policy file). The task S_0 also sets up the necessary system parameters and tasks used by $ATME$.

The other task that $ATME$ employs is the *execution monitor*, which is spawned by S_0 and communicates with user tasks at runtime to accept requests (from user tasks) and provide responses (to the user task) in order to handle conditional task execution.

The execution monitor exists until the completion of the user *exit* task (task E in the example). The interaction between the user task and the execution monitor is implemented by $ATME$ through the passing of "events" (Section 5.4.2) and is entirely transparent to the user.

5.4.2 Execution Events

This section describes all events transmitted to and from the *execution monitor* task. Events from the user task to the *execution monitor* include the detection of whether one of its child tasks has been spawned, whether a parent task has transmitted data, and whether it is this task's turn to commence execution on the allocated processor. Events also include the detection of the state of a user task (spawned or not spawned) and data communication (transmitted or not) between user tasks, and the termination of a task's execution.

The responses from the *execution monitor* task to the user task include the *task identification number* of a user task, the communication state between user tasks, and the execution start signal of an active task (i.e., whether this task has commenced its execution or not).

The events and responses between the *execution monitor* task and the user task are elaborated as follows. The *execution monitor* is abbreviated as EM below.

- Event *ev1_detchdspawn*: sent from a parent task to the EM task, when the parent task intends to spawn its child task. This event detects whether or not the child task has been spawned by its other parent tasks. If the child task is being spawned, or such information is unavailable, the EM holds this request until either one of the two signals can be determined: "not-spawned" or the *task*

identification number (allocated by PVM indicating that the task has already been spawned, by one of the other parent tasks).

- Event *ev1_detparsend*: sent from a receiving (child) task to the EM, detecting whether or not its parent task has transmitted a message. The response from EM is either "sent" (i.e., the data has already been transmitted.) or "never-sent" (i.e., the data is not to be transferred by the parent task, due to, say, the conditions associated with such transmission not being satisfied). The EM retains the request until a definitive answer can be obtained. With such a response, the receiving (child) task can then determine whether or not to suspend and wait for the data arrival.

- Event *ev1_detexecready*: sent from a task to the EM, detecting whether or not it is its turn to commence execution on its assigned processor. As previously stated, each task on the resident host has an execution commencement sequence, which is pre-determined by the scheduling policy (stored in the *scheduling policy file*). At runtime, such a sequence should not be violated, in order to achieve the system performance expected by the incorporated scheduling algorithm. The EM transmits this response if the task is the next to run on its assigned processor; otherwise, it holds this request until this task's turn arrives. With a definitive response from the EM, the task can then commence its computation.

- Event *ev1_spawn*: sent from a spawning (parent) task to inform the EM that it has spawned a child task, so that this child task will not be spawned repeatedly by its other parent tasks, unless required to do so by the application itself. Recall that in conditional parallel programming, a child task can be spawned by one (and only one) of its parent tasks. Such information is retained in the EM for

future consultation by other tasks. The *identification number* of the spawned child task is also transferred to the EM for later reference (say, the request sent by the *ev1_detchdspawn* event).

- Event *ev1_nospawn*: sent from a parent task to inform the EM that it will not spawn a certain child task owing to the fact that the associated conditions are not satisfied. If this child task is not spawned by any of its parent tasks, the EM marks this child task as "not-spawned" (i.e., not running) at runtime, so that future inquiries from other tasks regarding the status of this child task can be confirmed and all operations (statically defined) related to this child task can be ignored. The EM also checks the existing requests (retained inside the EM) regarding the status of this child task, and replies to the sender of the request.

  The EM detects the "not-spawned" child task via an integer counter, which is initialized as the total number of its parent tasks and reduced by one each time one of its parent tasks does not spawn it. A "not-spawned" task sends events to the EM, to indicate that no data transmission and task spawning is conducted inside it. Therefore, succeeding (child) tasks of the "not-spawned" task (as illustrated in the static task model) do not have to wait for data from this "not-spawned" parent task, which may otherwise have been sent by this task.

- Event *ev1_send*: transmitted from a parent task to inform the EM that it has sent data to another task (i.e., child task, identified by its *task identification number*). This information is later acquired by this child task, in order to determine whether or not the corresponding data reception operations are exercised at runtime.

- Event *ev1_nosend*: transmitted from a sending task to indicate to the EM that it does not send data to the specified child task. Such information is used to

tell the receiving task (statically-specified) not to wait for a message from this sending task.

- Event *evl_nextexec*: sent for a task to indicate to the EM that the current task has terminated its execution and that the processor is ready for the next task to commence execution. Therefore, a task, which previously sends the EM the event *evl_detexecready*, accepts a confirmed answer from the EM and commences its execution.

- Event *evl_texec*: sent from a task to inform the EM that a parent task attempts to spawn a child task which has already been spawned by one of its other parent tasks. This event is used to capture the execution probability between interconnected tasks.

- Event *evl_exit*: only sent by the *exit* user task (in the task model), informing the EM that the application program has completed so that $ATME$ can terminate gracefully.

The design and implementation detail of the execution monitor is discussed in Section 5.4.3. All events and responses are transmitted as messages between user tasks and the *execution monitor* task. The workflow of the above events is detailed in Section 5.5. All event messages and response messages share the same data structure, respectively. Such messages just contain a few integer values, indicating information such as *task identification number* or *event type*, as seen in Section 5.4.3. Therefore, the magnitude of such messages is relatively small, compared to the application-related data sets (which are normally large) transferred between user tasks.

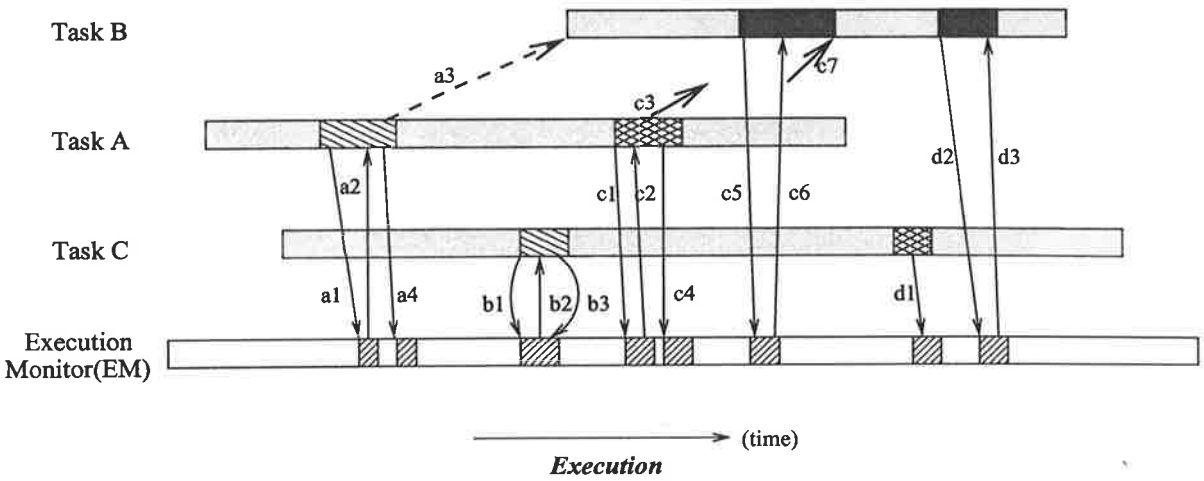
5.4.3 Design and Implementation of the EM

The *execution monitor* (EM) task, spawned by the $ATME$-generated task S_0, monitors the execution of user parallel tasks, and exists until the completion of the exit task in the parallel program. It manages a number of data structures which retain information such as the status of user tasks (e.g., spawned or not-spawned) and the communication between a pair of interrelated tasks. The main data structures employed by the execution monitor are listed in Appendix A. All events to and from the execution monitor task are encapsulated within the *in_msg* and *out_msg*, respectively, structures. Different events may have different associated parameters, which share the same space (defined by *union* in both structures). The *event_type* field in the *in_msg* structure indicates the event type, as presented in Section 5.4.2.

Waiting lists and queues created and maintained by the execution monitor are: *list_tidno*, *list_commmark*, *list_texec*, *wqueue_tidno*, *wqueue_commmark* and *wqueue_execready*. An explanation of these data structures is included in Appendix A. Pointers to each of these lists and queues are globally managed in the structure *pointers*.

The major processing of the execution monitor takes place within a loop: waiting for an event to come from the user task, dispatching the event to its corresponding function (implemented inside the execution monitor), responsing to the user task or holding the request, and then awaiting the next request from the user task.

The processing of each event received by the execution monitor basically follows the steps illustrated in Section 5.4.2. The full source code of the execution monitor is found in Appendix A.



Legend:

1. Processing within the task:

: processing on user tasks. : no processing on EM. : processing on EM.

: task spawn. : data transmission. : data reception.

2: Message passing operations between tasks:

: spawn operation. : communication between user tasks and EM.

: data transmission. [a-d][1-5]: sequence of operations.

Figure 5.11. Processing between the user task and the execution monitor.

5.5 Implementation of *ATME* Primitives

This section describes the processing of three major *ATME* primitives, i.e., *tme_spawn*, *tme_send* and *tme_recv*, in the following three sub-sections: Sections 5.5.1 through 5.5.3. The implementation details of the *ATME* primitives are illustrated via a task model as shown in Figure 5.10, with the focus on three tasks A, B and C.

Figure 5.11 illustrates the communication between user tasks (A, B and C) and the *execution monitor* task. This figure depicts views from *XPVM* [69] on which the *ATME*-preprocessed program is executed. The communication pattern between the three tasks (namely A, B and C), as illustrated in Figure 5.11. It is supposed that

task B is spawned by either A or C; data is actually transmitted between A and B but not between C and B.

In Figure 5.11, each task which is involved in the execution is represented by horizontal line. The length of the line illustrates the execution span of the corresponding task. The fill-in background of the line describes the kind of processing within the task (say, task spawn or data transmission), as indicated by the *Legend* in the figure.

Message-passing between an $ATME$ task and the execution monitor, as well as between any two $ATME$ tasks, is depicted by an arrow in Figure 5.11. All the events are time-stamped and ordered accordingly. Different types of arrows represent different kinds of message-passing operations, as presented in the *Legend* in the figure. Each message-passing operation is marked by a two-charactered label, for instance, $a1$. The letter in the label distinguishes between communication events. The digit in the label represents the sequence of events within each operation.

As observed in Figure 5.11, whenever a user task intends to invoke a message-passing operation or to spawn another task, the corresponding $ATME$ task needs to communicate with the *execution monitor* before and after the operation. Only after acquiring a definitive response from the execution monitor task does the task go on to its next statement. This can be seen from the following discussion.

5.5.1 Processing of *tme_spawn*

This section takes an example of a task-spawn operation and illustrates the processing of the $ATME$ *tme_spawn* primitive. The processing is undertaken by $ATME$, and is completely transparent to the programmer. Labels (such as "$a1$") in each step listed below refer to those marked in Figure 5.11.

Concentrating on the three tasks A, C and B of Figure 5.10, suppose both tasks A and C may spawn B at different times when the program executes, however, B can not be spawned by both A and C at runtime (presuming this is required by the application). As seen, communication between task A and EM, as well as between task C and EM can guarantee that B is uniquely spawned by either A or C. The following processing depicts the communication between $ATME$ tasks (A and C) and EM.

1. When one task (A or C in the example) intends to spawn another task (task B), the parent task sends the event *ev1_detchdspawn* to the EM task (Label $a1$ or $b1$, respectively), to detect the state of the to-be-spawned child task B. The parent task then waits for the response from the EM task.

   There are two possible responses: the child task has not been spawned by any of its parent tasks (marked as "not-spawned") or the child task has been spawned (recognized by the *task identification number*, i.e., *tidno* of the child task). The given example assumes that task A attempts to spawn task B, before task C intends to do so. Therefore, it can be realized that task A will have the "not-spawned" response from the EM, while task C will have the "tidno" response from the EM (since at that time task B has already spawned by task A).

   Depending on different responses from the EM task, $ATME$ employs the following two approaches, respectively, to manage the execution.

2. If the response from EM is "not-spawned" (task A has this response, since B does not exist when A intends to spawn B, as shown by step $a2$ in Figure 5.11), then $ATME$ arranges the spawn operation, within task A, of the child task via the PVM primitive *pvm_spawn* (step $a3$). In addition, task A sends EM the

"task identification number" of the newly-spawned task (i.e., B) (step $a4$) for later consultation (by task C in this example).

3. If the response is a "task identification number" (task C has this response, since B exists when C tries to spawn B), as referred to by step $b2$ in Figure 5.11, then this parent task (C) skips its task spawn operation, and sends EM an event *ev1_texec* to indicate that the communication does occurs between this pair of tasks, i.e., C and B (step $b3$).

As seen, $ATME$ realizes the conditional task spawn support through several interactions between user tasks and the *execution monitor* task. It is the $ATME$ environment, not the programmer, that retains all the runtime information, fulfills all these interactions and makes sure that task B is spawned by one, and only one, of its parent tasks.

Appendix B presents an example of a user-provided parallel task, and its translated $ATME$ task. The example includes all three task runtime operations: conditional task spawn, conditional data transmission and conditional data reception.

5.5.2 Processing of *tme_send*

Conditional data transmission in conditional parallel programming is complex in comparison to its PVM point-to-point data transmission counterpart, in that it may be involved in operations other than mere data transmission. As stated in Section 5.2.2, it is suggested that all parent tasks in the conditional parallel program are responsible for the spawning of its child task(s), due to the unpredicatable nature of the conditional branches associated with the task spawn operation. In this situation, the target child task can be guaranteed to exist when the communication is triggered, since the

communication between each pair of tasks is always preceded by a spawn operation of the child task. However, this may not be true as the actual program is developed. That is to say, it is likely that a parent task realizes that the child (receiving) task may not exist when it prepares to transmit data. These additional issues have to be addressed in conditional data transmission.

There are two situations (explained below) which must be dealt with in conditional data transmission, depending on whether the data transmission actually takes place or not at runtime. If data is actually transferred between the *sending* and *receiving* task, say A and B in Figure 5.10, the following occurs:

1. The sending task (A) sends an event *ev1_detchdspawn* to EM in order to obtain the *task identification number* of the receiving task (B), as indicated by step $c1$ in Figure 5.11.

   According to the response returned from the EM, one of the following two steps is taken.

2. If the response from the EM task is "not-spawned", then $ATME$ issues a task spawn operation inside the parent task for the receiving task, following the process stated in Section 5.5.1. This is not illustrated in Figure 5.11, since, in this example, it is assumed that the receiving task (B) has already been spawned by the sending task (A) at the time when data transmission is processed.

3. If the response from the EM task is the "task identification number" (step $c2$ in Figure 5.11), i.e., the receiving task B has already been spawned, then the sending task (A) issues a PVM send primitive (by *pvm_send*) in which the destination is recognized by the "task identification number" just obtained from the EM (representing the receiving task) (step $c3$). In addition, the sending task (A)

also informs the EM of the occurrence of such communication (step $c4$). Note that step $c3$ is undertaken regardless of the state of task B, this is referred to as "asynchronous blocking data tranmission" in PVM (Section 5.3.3).

4. The remaining part of the sending operation, i.e., the data reception, is realized by the receiving task (i.e., step $c5$ to step $c7$), which is discussed in Section 5.5.3.

Another situation in conditional data transmission occurs when there is no data communication between two interrelated tasks, due to a false condition associated with the communication (between C and B in the example). In this case, the following steps are followed:

1. The parent task (C) sends an event *ev1_nosend* to EM, informing it that no data is to be communicated between the two interrelated tasks C and B. This step is marked by $d1$ in Figure 5.11. This step actually sets an indicator inside the EM task, for any later inquiry from the receiving task B.

2. The remaining part of the sending operation, i.e., data reception, though actually not taking place, is processed by the child task B (i.e., step $d2$ to step $d3$), which is discussed in Section 5.5.3.

As seen, conditional data transmission is not simply involved in transmitting the data, but may involve the spawning of a task as well. $ATME$ provides the primitive *tme_send*, which frees the programmer from considering these extra issues.

Appendix B illustrates the original user-written code regarding (conditional) data transmission and its translated $ATME$ code.

5.5.3 Processing of *tme_recv*

Data transmission and data reception are an inseparable pair of operations in terms of message-passing. Data reception may also be conditional. Depending on whether or not there is any data sent from the parent task, the child (receiving) task can decide whether or not to suspend itself and wait for the arrival of the data. The following steps in conditional data reception are undertaken:

1. The child (receiving) task (B in the example) checks with the EM through the event *ev1_detparsend* to see whether there is any message transmitted from a sending task, i.e., task B's parent tasks A and C. This is marked by step $c5$ and $d2$, respectively, in Figure 5.11.

   There are two possible answers from the EM task, and $ATME$ takes one of the following directions, accordingly.

2. If there is a message from the sending task (i.e., A, indicated by the step $c6$), then the receiving task B is blocked waiting for the data to arrive in the message receiving buffer of B (step $c7$). Here it is assumed that the identification of the message matches both the sending and the receiving tasks. In other words, it is the message required by this receive operation.

   The receiving task also arranges related operations such as unpacking data to decompose the message into its original format to be used within the task itself. The data transmission and reception (between A and B in the example) is then regarded as complete.

3. If, on the other hand, there is no message sent from the parent task (C in the example), as indicated by step $d3$, the receiving task (B) skips the receiving

and all related (such as unpacking data) operations, and continues with its computation.

Appendix B also presents the implementation of the conditional data reception in a user task.

As observed from the implementation of the three $ATME$ major primitives (Sections 5.5.1 through 5.5.3 above), the $ATME$-introduced *execution monitor* task realizes $ATME$ support for conditional parallel programming and the automation of the conditional task scheduling process. The application programmer, equipped with the $ATME$ primitives, is largely freed from considering new issues which arise in conditional parallel programming. Furthermore, system and program performance is enhanced due to the efficient task scheduling policy which is generated according to special requirements of parallel tasks and processors. It has been experimentally shown that CET, which is employed by $ATME$ as its scheduling algorithm, is generally superior to the round-robin algorithm adopted by the PVM tool.

Chapter 6

ATME: A Tool For Conditional Parallel Programming

This chapter presents the design and implementation of a programming environment, named *ATME*. The contribution of *ATME* is built upon the research discussed in Chapters 3, 4 and 5. This chapter discusses in depth the functional components of the *ATME* environment, and highlights the cooperation of the *ATME* components to realize support for parallel programmers. Section 6.1 presents the framework of *ATME* and the work flow between the *ATME* components. The construction of the processor model (i.e., target machine topology) is discussed in Section 6.2. Sections 6.3 and 6.4 examine the preprocessing and the analysis of the user program, in order to ensure it executes on the *PVM* platform and captures task runtime information. Such information is employed in the incremental construction of the task model discussed in Section 6.5. The generated task model and the processor model are two essential inputs into the scheduling algorithm which produces a scheduling policy prior to program execution. This aspect of the work is discussed in Section 6.6. Section 6.7 focuses on the collection of the runtime task information and its retention in the program

databases (for use in task model construction). Section 6.8 describes the remaining components in *ATME*, which includes post-execution analysis and report generation.

6.1 *ATME* Framework

This section presents the framework of *ATME* and briefly states the functionality of each *ATME* component and the relationships between them.

ATME is developed atop the *PVM* platform [69]. As required by *PVM*, an application parallel program is partitioned into parallel tasks, each of which realizes some sub-functionality of the application problem. Tasks are physically mapped onto the target machine connected by *PVM* before they commence their execution. Tasks on different processors (computers) in the target machine can run in parallel, unless synchronization is required between tasks.

Given a virtual parallel machine, an *ATME* user remains unaware of the technical details of the underlying parallel and distributed system, and simply assumes that the target machine is composed of fully-connected identical processors. This allows the underlying platform to be modified with minimal interruption to the application programmer, and ensures that *ATME* achieves maximum portability. Furthermore, with the availability of a set of *ATME* runtime primitives, the programmer is relieved of the need to be concerned with subtle characteristics of the parallel system, and more significantly, new issues related to conditional parallel programming, as stated in Section 5.2.

Figure 6.1 provides the framework of *ATME*. *ATME* takes as input a *processor topology specification* and *user-defined parallel tasks*. The *target machine description* component presents the user with a general interface to specify the available processors

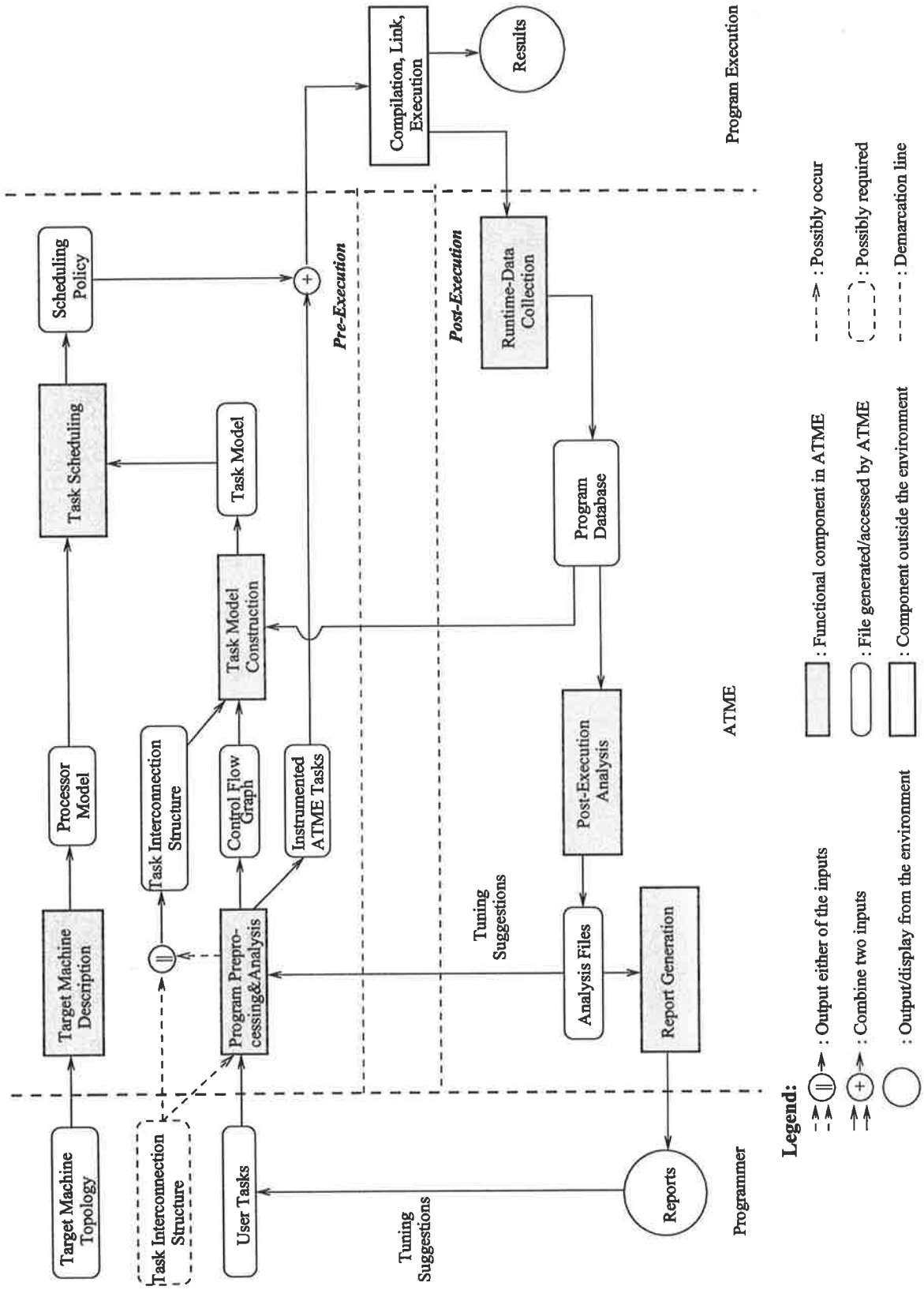


Figure 6.1. Framework of *ATME*.

and the topology of the underlying parallel and distributed system. It accepts as user input a processor topology specification, and generates the processor model for use by other *ATME* components.

The user-provided parallel program is analyzed, preprocessed and instrumented by the *program preprocessing and analysis* component in order to enable it to execute on the *PVM* platform and capture appropriate information at runtime. Such information is used to construct the task model of the program (as required by the conditional scheduling algorithm, i.e., *CET* and *PET*, employed by *ATME*). Currently, *ATME* supports parallel programming in C. *ATME* provides explicit support, through *ATME* primitives, for conditional parallel programming.

Task runtime information in every program execution is retained within *ATME* (discussed shortly). Based on the information captured in previous executions, the *task model construction* component predicts the actual task model prior to the forthcoming program execution. With the task model from the *task model construction* component and the processor model from the *target machine description* component, the *task scheduling* component, which mainly incorporates a scheduling algorithm, statically generates a policy by which the user tasks are distributed onto the underlying processors. The task model and the processor model employed in this thesis are adopted from the large number of existing scheduling algorithms. Therefore, other scheduling algorithms can be easily plugged into *ATME*.

At runtime, the *runtime data collection* component collects traces produced by the instrumented tasks. Trace information is stored, and after the execution completes, dumped into *program databases* which are used as input by the *task model construction* to predict the task model for the next execution. The *post-execution analysis* and *report generation* provides analysis, various reports and tuning suggestions to the user,

as well as to *ATME* itself, with the aim of enhancing system performance.

As observed, a cycle exists in the *ATME* environment, as seen in Figure 6.1: commencing with the *task model construction*, through *task scheduling*, *runtime data collection* and back to the *task model construction*. This process makes *ATME* an adaptive environment in that the task model offered to the scheduling algorithm is incrementally established to gradually reflect changes in task behaviour, with the expectation of improving the distribution strategy on the basis of past usage patterns of the application. This process depicts the strategy employed in this thesis to tackle task scheduling issues for conditional parallel programming. Accurate estimation of task attributes can be obtained for relatively stable usage patterns and consequently admits an improvement in execution efficiency.

6.2 Target Machine Description

With the user-specified processor topology, *ATME* establishes a processor model of the target machine, which is required by the scheduling algorithm employed in *ATME*. *ATME* also presents the user with a *virtual parallel machine* (through *PVM*) so that the user is unaware of the architectural and performance details of processors and networks. This is realized by the *target machine description* component of *ATME*.

The *target machine description* component allows the user to specify the available processors, processor interconnections and their associated attributes such as processing speed and network bandwidth. It is realized that different scheduling algorithms may emphasize different processor and network modeling factors. For instance, with respect to processor network modeling, most of the task scheduling algorithms consider the network bandwidth (i.e., data transfer rate) [93, 115, 151, 180],

```
/*** File Structure: Processor ***/
struct processor   {
  int           procidxno;              // Processor index number.
  char          procnm[NAMELEN];        // Processor name.
  double        psr;                    // Processing speed.
  double        inittime;               // Message-sending overhead.
  int           pconnum;                // Number of outgoing links.
  struct proccon  {
          int             procidxno;
          int             linkidxno;    // See "Network Link" below.
  }             proccons[MAXPROCNUM];
};


/*** File Structure: Network Link ***/
struct netlink   {
  int           linkidxno;              // Processor link index number.
  int           procidxno1;             // One end of the link.
  int           procidxno2;             // The other end of the link.
  char          linknm[NAMELEN];        // Link name.
  double        bandwidth;              // Date transfer rate.
  double        contlimit;              // Contention limit.
  double        prolong;                // Message-sending overhead.
  int           busy;
  double        occupysize;
};
```

Figure 6.2. The target machine description file.

while others may be also concerned with network contention and initiation of message passing [52]. Therefore, the *target machine description* provides a general interface which covers processor attributes commonly required by scheduling algorithms.

The output of the *target machine description* is a processor model which is regarded as one of inputs to the *task scheduling* component. The processor model adopted by *ATME* is formally defined in Section 3.1.

The processor model is kept in two files: the *processor* file and the *network link* file, the structure of which is given in Figure 6.2. The design of the processor model

provides the facility to describe a heterogeneous system. The research undertaken in this thesis on homogeneous systems can be extended to heterogeneous systems in a straight-forward manner. Futhermore, the processor model retains more information than it is currently used by *ATME*'s incorporated algorithms (*CET* and *PET*). These factors, such as message-sending overhead and network contention limit, are intended to be used by scheduling algorithms which may be embedded in the future.

Apart from the construction of the processor model for the target distributed system, the *target machine description* component establishes a virtual parallel machine, through underlying *PVM* mechanisms, by linking all participating processors via existing networks. *ATME* (through *PVM*) starts a daemon process on each available processor to handle the interprocessor communication.

With respect to a specific application, the underlying target machine, on which the application program executes, can generally be assumed to be stable. That is to say, the parallel and distributed system is not likely to change along with the program. Therefore, the processor model of the target machine, once established, does not need to be reconstructed each time an application program is executed. The *target machine description* provides the physical foundation on which the user program is executed, and makes the actual system transparent to the user. A user may be unaware of the physical characteristics of processors and communication networks, while concentrating merely on the parallel program and its algorithms.

6.3 Program Preprocessing

This section and the following section focuses on the processing of the parallel program, which is realized by the *ATME program preprocessing and analysis* component.

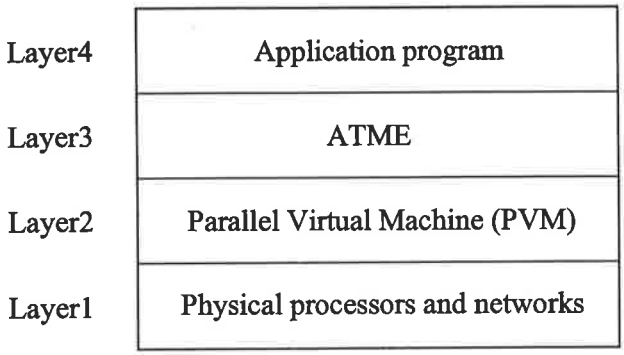


Figure 6.3. Layers of software.

In order to implement a software package efficiently and productively and to enhance its reusability and portability, it is desirable to build software in layers. Figure 6.3 gives an example of a software hierarchy within a parallel and distributed system. The application program sits on top of the $ATME$ environment, which itself is located above the PVM layer. Software layers beneath PVM manage the physical parallel and distributed system. The application programmer is equipped with services provided by lower layers while not being concerned about implementation details underneath. The user-provided program (with $ATME$ primitives) is first translated into the corresponding PVM program before it is actually submitted for execution on the underlying target machine. Such transformation is automatically undertaken by the $ATME$ *program preprocessing and analysis* component.

This section discusses the issues involved in program preprocessing, and defers the program analysis to Section 6.4. In $ATME$, the preprocessing of the parallel program is involved with four major issues. Firstly, it ties the user-provided parallel tasks to two $ATME$-specific tasks: the start task S_0 and the execution monitor task. The task S_0 establishes appropriate parameters, loads files (such as the task interconnection file, processor topology file and task scheduling policy file), triggers the execution monitor task and distributes the first (start) task in the user program. The execution monitor

task is mainly responsible for the runtime management of task and task communication information. The design and implementation details about the execution monitor task and the start task S_0 are presented in Chapter 5.

Secondly, the program preprocessing in $ATME$ interprets the $ATME$ runtime primitives (incorporated in the user-provided tasks). The output of such preprocessing is C code mixed with PVM primitives. The preprocessing also includes the interaction between the translated PVM tasks and the execution monitor task, which realizes the support for conditional task spawn and conditional message-passing at runtime. The workflow of the three major $ATME$ primitives (i.e., task spawn, data transmission and data reception) is illustrated in Section 5.5. An example of a user-provided parallel task, and its corresponding $ATME$-translated task (i.e., PVM task), is presented in Appendix B.

Thirdly, the $ATME$ preprocessing of the program inserts new source code into the user-provided program, for the management of execution. The scheduling policy generated by other $ATME$ components specifies the task distribution strategy and the execution order of tasks assigned to the same processor. With $ATME$, the application programmer can completely ignore issues involved in task scheduling. It is $ATME$ that generates the scheduling policy, and dynamically accesses the policy file to control the task execution. This is implemented through the functions *getproc* and *det_execready* embedded within $ATME$. An example can be found in Appendix B.

Finally, the program preprocessing in $ATME$ extracts the task interconnection structure of the parallel program that portrays the precedence relationships between tasks. The interconnection structure is a vital component of the task model. Such a structure can be obtained by detecting task runtime operations within the program. A "spawn" or "send" operation in the task indicates a precedence relationship in the

task model. The programmer can provide his/her own task interconnection file, which may be the product of other program design tools, such as HeNCE [15].

From Appendix B, it can be observed that there is outstanding difference, in terms of program size, between the $ATME$ program (written by the application programmer) and the corresponding PVM program (generated by $ATME$ from the user program). For instance, when sending data from task A to task B, about 10 lines of source code needed to be written by the application programmer. In order to make the user-provided task physically execute on the underlying PVM virtual machine, such code is expanded into more than 40 lines. This excludes the preparatory work undertaken in the beginning of the program. Furthermore, the extended code also includes a number of $ATME$-developed functions, such as *det_parsend* and *put_nextexec*, which handle the communication between the user task and the execution monitor in $ATME$. This means that a significant amount of work dealing with conditional data transmission is hidden inside provided $ATME$ library routines. All such processing would be imposed onto the programmer, if it were not for $ATME$.

6.4 Program Analysis

This section studies the analysis of the user supplied application programs. The aim of such analysis is to insert probes into the program so as to capture task runtime information, convert it into task attributes and retain them as program profiles, upon which the scheduling policy for the conditional parallel program can be produced.

The types of task attributes concerned with scheduling tasks varies from one algorithm to the next. For example, the scheduling algorithm CET (Section 3.6) takes three factors into account, namely, task computation time, task communication

time and execution probability, while the algorithm PET (Section 4.4) considers the preemption start point, instead of the execution probability used in CET. This section concentrates on the acquisition of these four attributes.

The strategy to undertake program analysis is outlined as follows. For each parallel task, a control flow graph is generated to describe the relationship between statements within the task: related statements are gathered into a node in the control flow graph while an edge in the graph represents the control dependence between nodes. Then, the task is instrumented with probes in order to gather the runtime information. Each node is regarded as an atomic instrumentation unit. Consequently, at runtime, probes generate node attribute values which are collected by $ATME$'s *runtime data collection* component. Such node attributes are summarized into task attributes, for the use of the task model construction component in $ATME$.

Section 6.4.1 illustrates the mechanism to build the control flow graph for an $ATME$ task. Task instrumentation is discussed in Section 6.4.2, which presents the type of probes inserted into $ATME$ tasks. Methods to reduce the number of probes instrumented in the program are also studied, with the aim of eliminating the impact of probe perturbation on task behaviour.

6.4.1 Control Flow Graph

The execution of a task is encapsulated in a *control flow graph* [168] (abbreviated as CFG), in which each node represents a source code fragment within the task, and each edge describes the control dependence between the task fragments. A formal definition of the CFG is given below. The aim of introducing the control flow graph is to determine the value of task attributes (including task computation time, communication data time, execution probability and preemption start point)

and provides a basis for task instrumentation. The use of the control flow graph is widely employed in parallel processing [59, 105, 167]. No further discussion on the control flow graph itself is developed here.

With the focus being the extraction of attributes of the parallel task, *ATME* presents a simplified control flow graph, which is formally defined as a n-tuple: $G = (N, L, N_s, N_e, A)$, where:

- N: the set of nodes, each of which is composed of a source code fragment in the corresponding task. The partitioning of nodes within a task is based on the node type, as discussed below. A node is regarded as an atomic unit from the perspective of instrumentation (as seen in Section 6.4.2). Virtual nodes are employed to describe the branch and loop structure.

- L: the set of directed links between nodes. It portrays precedence (control flow) relationships between nodes.

- N_s: the start node of the control flow graph. With no loss of generality, it is assumed that each task has a unique start node.

- N_e: the exit node of the control flow graph. It is assumed that there is only one exit node for each task.

- A: the set of node attributes. Each node is associated with *node attributes* (elaborated below) which are either extracted through static program analysis or obtained through runtime execution profiling.

An example of a task and its corresponding control flow graph are shown in Figure 6.4 (a) and (b), respectively. Nodes are delimited by rectangles, while directed arrows illustrate the control flow between nodes. In Figure 6.4 (a), three typical

```
# include <.....>

main()
{
  [Variable Declarations]

  [Code 1: sequential code fragment]
  while  (condition a)
    {
      if  (condition b)
        {
          [Code 2: sequential code fragment]
        }
      else
        {
          [Code 3: sequential code fragment]
        }
      [Code 4: sequential code fragment]
    }
  [Code 5: sequential code fragment]
}
```

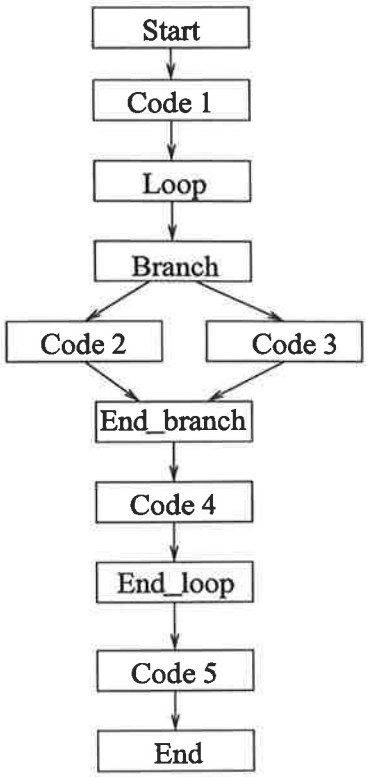


Figure 6.4. An example of (a) a task and (b) its corresponding control flow graph.

kinds of programming statements are illustrated, that is, the sequential processing statement, the loop statement and the conditional branch statement. The mapping of the statement fragment in Figure 6.4 (a) and the corresponding node in Figure 6.4 (b) is manifested by the code label, shown as *Code*1 and *Code*2 etc. *Loop* and *Branch* nodes represent the opening of the loop and branch structures in Figure 6.4 (a), respectively. *End_loop* and *End_branch* are virtual nodes introduced into the control flow graph, to indicate the closing of the loop and branch structure, accordingly.

The data structure representing a node is illustrated in Figure 6.5. As seen, each node in the control flow graph is uniquely identified by a number *blkno*. Basically, the structure aims to portray node attributes (such as the node execution time *etime*) and node inter-relationships (such as *nextblkno*[]). "Related node" items in Figure 6.5 are designed for indirect reference between nodes, with the purpose of reducing the number

```
/*** A node in the control flow graph ***/
struct node   {
  int           blkno;                  // Node identification number.
  int           blktype;                // Node type.
  int           levelno;                // Level number.
  int           nextblknum;             // Number of successive nodes.
  int           nextblkno[MAXNEXTBLKNUM]; // All successive nodes.
  int           prevblknum;             // Number of preceding nodes.
  int           prevblkno[MAXNEXTBLKNUM]; // All preceding nodes.
  int           stime;                  // Related node (for node's
                                        // Execution start time).
  int           ind_stime;              // Related node indicator.
  int           ftime;                  // Node finish time.
  int           etime;                  // Actual node execution time.
  int           count;                  // Related node (for node's
                                        // Execution frequency).
  int           ind_count;              // Related node indicator.
  int           ecount;                 // Actual node repetition count.
};
```

Figure 6.5. The data structure of a CFG node.

of instrumentation probes, as discussed in Section 6.4.2.2.

Major node attributes defined in Figure 6.5 are elaborated as follows. They are required by the calculation of task attributes, which play a crucial role in the construction of the task model. These node attributes include node type, node level, node execution time and node execution frequency.

- *nodetype*: the type of the node.

  Each statement within a task has associated with it a *type* which can be *assign* (assignment statements), *branch* (selection statements), *loop* (repetition statements), *comment* (comments), opening and closing brackets (which indicate the beginning and the end of a structured construct). The control flow graph illustrates the most common and generic constructs in programming languages.

Each node of the graph contains consecutive statements of the same kind (except for embedded branch and loop statements) in the task. Nodes are typed, according to the constituent statements. Three generic node types are considered here: the *basic* node (usually labelled with a "node name" such as *Code*1 in Figure 6.4 (b)) represents a sequence of assignment statements where the control flow enters at the beginning and exits at the end without the possibility of branching; the *Branch* and *Loop* nodes represent the beginning of the selective and iterative constructs respectively. Data declarations are assumed not to require any CPU time, thus there are no particular node types dealing with them (recall that nodes in the graph are used to determine task runtime attributes).

Several virtual nodes are introduced to a task's CFG: the *Start* and *End* nodes represent the beginning and end of a task respectively, while the *End_branch* and *End_loop* nodes denote the end of the selective and iterative constructs respectively. Such virtual nodes allow embedded structures (such as an embedded *Branch* or *Loop*) to be easily distinguished in the graph. Furthermore, there are no backward edges in the graph for a repetitive structure. Its start and finish is distinguished by the pair of *Loop* and *End_loop* nodes. Therefore, the control flow graph is easily understood and implemented.

- *level*: each node is assigned a *level* number to indicate its level of nesting within the graph. The *Start* node and the *End* node of the control flow graph have a level number of 0. Entering a conditional branch or a loop increases the level by 1, while the end of the branch or loop decreases the level by 1. Note that nodes of the same level number (such as *Code1* and *Code5* in Figure 6.4(b)) may have the same repetition count at runtime. That is to say, they may be visited

exactly the same number of times. Such properties can be exploited to eliminate unnecessary probes inserted in the task, as discussed in Section 6.4.2.2.

- *etime*: the execution duration of the node. It is also simply called "execution time". *etime* is calculated as the difference between the start time (*stime*) and the finish time (*ftime*) of the node.

- *ecount*: the execution frequency of the node at runtime, i.e., the number of times this node is visited when its encapsulating task is executed. This attribute, along with the attribute *etime*, is gathered at runtime and profiled into the trace files at the end of task execution.

The extraction of the control flow graph from the task's source code is performed as follows.

1. Initially, only the *Start* virtual node is placed in the control flow graph CFG. Set the current node *level* value to 0. Set the start node as the "last_visited" (LV) node. Start from the first statement in the task.

2. Depending on the *statement type* of the current statement in the task, different approaches are employed:

   - If the current statement is typed as *assign*, then amalgamate the subsequent statements of the same *statement type* and the same *level* value into a single node. Add this node in the CFG. A directed link is attached between the LV node and this node. Assign node attributes, such as *blkno* and *levelno*, to the newly-generated node. Note that such node attributes are initialized whenever a new node is generated and placed into the CFG. The new node is set as the LV node.

- If the current statement is typed as *branch*, then create a *Branch* node in the CFG with two outgoing directed links, for the two possible directions (named as *if* and *else* directions) of the conditional branch.

  Suppose both sub-blocks of the branch statement are delimited by brackets, even though it may include only one statement. This is for simplicity of discussion. Link this node to the LV node. Set this node as the LV node.

- If the current statement is typed as *loop*, then create a *Loop* node in the CFG with an outgoing link. It is also assumed that statements inside a loop are bracketed. Link this node to the LV node. Set this node as the LV node.

- If the current statement is an opening bracket (which indicates the beginning of either a selective or a repetitive structure), increase the current node *level* value by 1.

- If the current statement is a closing bracket of an *if* direction sub-block (i.e. its statements are executed when the conditions associated with the previous *branch* statement are satisfied), then decrease the *level* value by 1, and start dealing with the *else* alternative of the same *branch* statement. The current LV node is set as the "branch" node.

- If the current statement is a closing bracket of an *else* alternative of a *branch* statement, then decrease the node *level* value by 1, create an *End_branch* node and connect it with two nodes which are the branch's two sub-blocks. Set the *End_branch* node as the LV node.

- If the current statement is a closing bracket of the previous *loop* statement, then decrease the *level* value by 1, put an *End_loop* node in the CFG, link

it to the LV node. Set this node as the LV node.

- If the current statement is a closing bracket representing the end of a task, then an *End* node is placed in the CFG, link it to the LV node. The generation of the task's control flow graph completes. The node *level* value becomes 0 again.

3. The above step is repeated until the end of the source code is encountered.

On the whole, the control flow graph discussed here deals with the basic constructs of a programming language. Specifically, these constructs include assignment, conditional branch and repetition. The control flow graph does not take into account procedure/function calls and recursive procedures. To tackle these more complex situations, additional nodes and relationships are required in the control flow graph. This is left to future work.

The control flow graph of a task constitutes a foundation for probe instrumentation and task attribute calculation, while the actual control flow between nodes of this task is of little interest. Hence, no attempt is made to introduce a complex graph structure which aims to describe the complete control dependencies within the task, as done in [59, 105, 167].

6.4.2 Task Instrumentation

Based on the control flow graph, the task is instrumented to capture values of node attributes at runtime. Such task runtime information is dumped into trace files, and retained permanently as execution history.

Section 6.4.2.1 discusses the different kinds of probes inserted into the user program. Since the instrumented probes affect task behaviour and prolong task execution time,

Section 6.4.2.2 examines strategies to eliminate the probe perturbation.

6.4.2.1 Instrumented Probes

Task instrumentation can be realized on different levels. On one hand, probe insertion can be conducted at the compiler or operating system level, through modifying the compiler itself or incorporating additional system calls. Thus, there is no need to develop additional environments or tools to accomplish task instrumentation. However, such an approach affects all parallel programs and requires the modification of system software.

On the other hand, probe insertion can also be undertaken at the application program level, an approach which this thesis adopts. Whichever approach is followed, the principles are the same: the process of probe insertion should be transparent to the application programmer as much as possible; and the inserted probes should impact on the task itself as little as possible. This section studies probe insertion in $ATME$, and Section 6.4.2.2 discusses approaches to minimize the impact of the probes.

In order to determine the values of task attributes for each executable user task, the following information needs to be collected. The determination of task attributes is based on the control flow graph of the task, as discussed in Section 6.7.2. The data collected includes:

- the computation time of each node in the control flow graph (in order to calculate the task attribute of *computation time* for the corresponding task),
- the volume of data transmitted to dependent child tasks (to obtain the task attribute of *communication time*),
- the *execution probability* between potentially communicating tasks,

- the *preemption start point* between parent and child tasks, and
- task precedence relationships of the parallel program.

Correspondingly, the following types of probes are employed to extract values relating to node and task attributes. These are:

- *timing probe*:

  This probe is set to gather the execution start and finish time of nodes within a task's control flow graph. The timing probe is inserted at the beginning and at the end of each composite node of the task. For a *basic* node, if the node is processed more than once during program execution, then the start and finish time of such a node is collected just once. The assumption made is that the execution time of a basic node in different visits is the same throughout the same program execution. This strategy is intended to reduce the probe impact on task execution by reducing the amount of CPU time used. In addition, after a data-transmission operation within the task, the *timing* probe can also be used to detect the preemption start point between a parent and its dependent child task.

- *counting probe*:

  The counting probe is inserted to determine the execution frequency of the node in the control flow graph. With the information produced by the *timing* probe, the total execution time of a node within the task in a program execution can then be determined.

- *size probe*:

  The size probe aims to detect the magnitude of data transferred between tasks. The data size is measured in data packet units, the size of which is defined by the user according to the actual network. In *ATME*, the size probe is inserted after each data packing or marshalling primitive in the task. All data sent to the same target task from the parent task is accumulated by the same size probe. Therefore, such a size probe determines the magnitude of data transmitted (and, therefore, the communication time) between interconnected tasks. Furthermore, the size probe can also be used to capture information about whether there is any interaction between a parent and a dependent child task (i.e., the task attribute of *execution probability*).

- *destination probe*:

  This probe is utilized to build task precedence relationships if the task structure is not provided by the application programmer. For each data transmission primitive in the task, this probe records to which task data is transmitted. That is, the *destination* probe is inserted after the data-transmission primitive.

- *input parameter probe*:

  As stated in Section 3.5, the behaviour of a parallel task is assumed to be determined by the input parameter applied to the task when it is spawned. *ATME* provides a switch, named *incps*, which allows the user (programmer) to set the value of "on" or "off", according to whether, in reality, the task behaviour is uniquely determined by the input parameter. If the *incps* switch is set as "on", then the *input parameter probe* is inserted (i.e., the value of the input parameter in a certain program execution is recorded); otherwise, no instrumentation regarding

the task input parameter is performed. The task model construction also depends on the value of the *incps* switch to determine its strategy in estimating the task attribute values, as seen in Section 6.5.

In *ATME*, probes are automatically inserted into user tasks at the application program level. Therefore, there is no necessity to modify an existing compiler or operating system. The inserted probes produce runtime information which is captured and stored for the purpose of constructing a task model representing a future execution of the program.

6.4.2.2 Probe Reduction

Along with the benefit brought about by the probes, the probes also incur additional runtime overhead by competing with the user-provided parallel tasks for the CPU time. Such probe perturbation is inevitable [106]. Various techniques have been proposed to minimize the impact of inserted probes, such as the logical clock [30], but none completely eradicates it. There exists a tradeoff between the advantage which task instrumentation provides and the overhead which it incurs. This section proposes several strategies to reduce the probe perturbation.

The control flow graph, proposed in this thesis as the basis for task instrumentation, is one of the strategies for reducing the magnitude of probes required to generate the desired data. In the graph, the "instrumentation grain" is extended from the individual *statement* to the *node* (a group of statements), which significantly reduces the number of probes inserted into user tasks.

Apart from the control flow graph, two other strategies aimed at reducing the number of probes inserted into tasks are presented in this section. The first strategy is to statically extract as much semantic information as possible regarding the task. Such

information includes the number of iterations in each loop, or a predetermination of the branch to be chosen in a selection statement. Therefore, related probes inserted in order to determine such information can be removed. The information may be obtained when analyzing and generating the control flow graph for each parallel task. However, in general, this kind of information can only be determined in very limited cases, since such runtime information is generally not available at compile time.

The second strategy is to make use of node relationships in the control flow graph. The following approaches are adopted in *ATME*:

- Share the probe between nodes:

  From the control flow graph, it can be observed that consecutive nodes (in terms of control flow) can partially share the *timing* probe, based on the fact that the finish time of a node is identical to the start time of the successive node. For instance, in Figure 6.4 (b), the start time of the *loop* node is equivalent to the finish time of node *code1*. With respect to the implementation, the *ind_stime* field (Figure 6.5) of the node *Loop* is set to 1, indicating that that the field *stime* references the node (i.e., *Code*1) where the value for its start time can be found. If the *ind_stime* is 0, then the *stime* field contains the actual execution commencement time of the node.

- Node reference:

  From the control flow graph, it can also be detected that nodes at the same *level* may have the same "execution frequency". Therefore, only one *counting* probe is required for such a group of nodes. For example, the *counting* probe for the node *code5* in Figure 6.4 (b) is unnecessary, since it has the same execution frequency as the node *code1*. As for the implementation in Figure 6.5, when the *ind_count*

is set as 1, the *count* field refers to the node with which it shares the execution frequency value; otherwise, the *count* field records the actual execution frequency.

- Attribute value deduction:

  There is no need to establish *counting* probes for each branch in a conditional structure. For a *branch* node, the control flow graph is constructed to include all possible branches of the conditional statement. Therefore, the sum of execution frequencies of all branches is equal to that of the *branch* node (which indicates the beginning of the selection statement). As a result, for an *if* statement which has at most two options, this approach may reduce the number of probes by 50% through attribute value deduction. Figure 6.4 (b) also illustrates this situation with the *branch* node and its two child nodes, *Code2* and *Code3*. The *counting* probe for the *Code3* node is not required.

Using the strategies presented above, the number of node attributes to be collected can be dramatically reduced. Attribute references between nodes can be achieved while generating the control flow graph via static program analysis.

Further probe reduction can be achieved in specific circumstances. For instance, if the target processors attributes (including architecture and processing load etc.) remain constant across different executions of the same parallel program, it is reasonable to assume that the execution time of a *basic* node (i.e., the processing time of such a node in once visit) is constant from one execution to the other. Therefore, in the initial run, information relating to the execution time of each *basic* node is collected; then, these probes can be removed in subsequent runs and future compilations.

6.5 Task Model Construction

The task model representing the parallel program is an important input to a scheduling algorithm. It reflects the precedence relationships between the parallel tasks. It also encapsulates the behaviour of tasks by way of its task attributes. Such information assists in producing an efficient scheduling policy by which the program can be executed with high performance. The construction of the task model for the application program is realized by *ATME*'s *task model construction* component.

ATME undertakes the task attribute estimation on the basis of the program's execution profile. The construction of the execution profile is discussed in Section 6.7.

The strategy to construct a task model is presented in Section 3.5. Continuous attributes such as *task computation time*, *communication time* and *preemption start point* adopt the linear regression model, as given in Section 3.5.2. Different approaches are employed to predict such task attributes, depending on the value of the *incps* switch (presented in Section 6.4.2.1). When the *incps* switch is set to "on", i.e., the task attributes are uniquely determined by the task's input parameter, the task model construction first estimates the *input parameter* for the task; then, using the estimated input parameter as the regressor, the above three task attributes are estimated. When the *incps* switch is turned "off", the above three attributes are directly predicted via the linear regression method, with the execution number as the regressor.

With regards to the *execution probability* attribute, this thesis proposes a finite state machine method to undertake the prediction, as discussed in Section 3.5.3. Simulation results presented in Chpater 7 indicate that *ATME* can predict the values with high accuracy, providing that usage patterns of the program are relatively stable across consecutive executions.

The component *task model construction* produces the complete task model (i.e., the task interconnection structure as well as task attributes), which is provided to the *task scheduling* component in order to generate a scheduling policy.

6.6 Task Scheduling

The *ATME task scheduling* component takes as input the processor model and the task model (generated from the *target machine description* and *task model construction*, respectively). It starts the incorporated scheduling algorithm, and produces a scheduling policy with the aim of optimizing the performance measure. The scheduling policy is retained in a file, which is accessed at runtime to distribute the tasks. The algorithm currently adopted in *ATME* can deal with conditional and preemptive task scheduling (note that deterministic task scheduling is just a special case of these two). The performance measure employed in *ATME* is *parallel execution time*, i.e., the schedule length of the entire parallel program.

The scheduling policy is illustrated by a chart: the horizontal axis is the list of all underlying available processors, while the vertical axis represents tasks allocated on each processor and the execution order of tasks assigned to the same host. The completion time of the *exit* task is the parallel execution time of the program. A graphical representation of the scheduling policy is found in Figure 1.1.

Within the *ATME* environment, the incorporated scheduling algorithm is a combination of the *CET* and *PET* algorithms, as proposed in Sections 3.6 and 4.4. As observed, *CET* and *PET* are similar: both algorithms aim to choose a schedulable task and an idle processor, so that the selected task can commence its execution on the chosen processor at the earliest possible time. The difference between *CET* and *PET*

```
struct  policy_file_format   {
  int           taskidxno;              // The task.
  int           procidxno;              // The allocated processor.
  int           execseqno;              // The execution order.
};
```

Figure 6.6. Data structures for the scheduling policy file.

is the different ways adopted to calculate the *earliest start time*, $S_1(t_i, p_j)$, of the task t_i on processor p_j (Section 3.6.2). Therefore, $ATME$ modifies the algorithm CET so that it can incorporate the processing of preemptive tasks scheduling, as follows:

$$\begin{aligned} S_1(t_i, p_j) &= \max\{(F(t_{i1}, p_{j1}) - U(t_{i1}, p_{j1}) * (1 - V(t_{i1}, t_i)) + \\ &\quad M(t_{i1}, t_i) * R(t_{i1}, t_i)) \mid \forall t_{i1} \prec t_i, D(t_{i1}) = p_{j1}, D(t_i) = p_j\} \\ S_2(t_i, p_j) &= \max\{S_1(t_i, p_j), A(p_j)\} \end{aligned}$$

where $S_2(t, p)$ represents the earliest start time of task t, considering the available time of its resident processor p.

The structure of the scheduling policy file is given in Figure 6.6. The policy file records the allocation of tasks onto processors, and the execution sequence of tasks resident on the same host. Such an $ATME$-generated policy file is accessed by the $ATME$ task spawn primitive *tme_spawn*(), to distribute tasks onto available processors. In this way, $ATME$ automates conditional task scheduling.

6.7 Runtime Data Collection

$ATME$ designs a functional component referred to as *runtime data collection* to capture and retain traces generated by the probes embedded in parallel tasks. This section examines issues addressed by this component.

It is node attributes of the control flow graph that are captured and stored in *trace files* at runtime. After execution, node attributes are used to calculate task attributes, which are retained for later use. The *ATME runtime data collection* component collects the runtime information relating to parallel tasks and processors, calculates the task attribute values, and stores such values into *program databases* as an execution profile.

Section 6.7.1 lists all trace files generated at runtime. The calculation of task attributes, based on the control flow graph and the captured information, is presented in Section 6.7.2. Section 6.7.3 illustrates the program databases employed in *ATME*.

6.7.1 Trace Files

When a task is executed at runtime, the inserted probes capture runtime data with regard to node attributes of parallel tasks, which are dumped into *trace files*. Corresponding to probes presented in Section 6.4.2.1, traces include the execution time of task segments, the volume of data communicated, the execution probability indicating whether or not the execution between two interconnected tasks actually takes place, and the preemption start point of data transmission operations. The trace file may also contain the information regarding the input parameter of the task. The following trace files are designed for the collection of node attributes in *ATME*:

- *traces1*:

  Each task has a *traces1* file, which captures the execution time and execution frequency of nodes in the control flow graph for that task. This file collects task runtime information which is used to calculate the *task computation time* attribute of the task model, as discussed in Section 6.7.2.

- *traces2*:

  Each task produces a *traces2* file. This file captures the volume of data communicated between this task and other interconnected tasks. Such information is used to calculate the *communication time* attribute of the task, since it is assumed that the communication volume is directly proportional to the communication time. Furthermore, the *traces2* file also notes the location within the source file of where the data communication takes place. The *preemption start point* between a parent task and its dependent child task can then be determined.

- *traces3*:

  All user tasks of the parallel program share a *traces3* file. The file records the spawn and attempt-to-spawn relationships between interrelated tasks in the program. Subsequently, the *execution probability* between tasks can be obtained from this file.

- *traces4*:

  When the *incps* switch is set as "on", this file gathers the value of the input parameter to the task. The collection of such information is used to predict the task model in subsequent program executions, as stated in Sections 3.5 and 6.5.

Trace files are by-products of program execution. After the program has been executed, trace files produced at runtime are transformed and dumped into corresponding "program databases" which profile the execution history of the parallel program.

6.7.2 Task Attribute Calculation

This section examines the calculation of task attributes, on the basis of attributes of nodes which are recorded in the trace files, as discussed in Section 6.7.1.

The basic attributes required to construct the task model include task computation time, task communication data time, execution probability and preemption start point between parent and child tasks. Among these task attributes, the *communication time* between interrelated tasks can be directly derived from the communication volume, which is presented in the file *traces2*. The *execution probability* is also directly obtained from the trace file *traces3*. Therefore, only *task computation time* and the *preemption start point* need to be calculated by referring to the corresponding control flow graph and trace files.

Armed with the trace files generated at runtime and the attribute references extracted from the static program analysis, the execution time and execution frequency of each node in the control flow graph can be achieved. The computation time of the task can then be calculated by:

$$U(i,p) = \sum_{j \in nodes_i} (etime_j * ecount_j)$$

where $U(i,p)$ is the computation time of the task i on processor p; $nodes_i$ is the set of nodes of task i's control flow graph; $etime_j$ and $ecount_j$ are the execution time and execution frequency, respectively, of node j in the graph.

With the availability of the task computation time, the preemption start point of its child task can then be achieved by:

$$PSP(i,j) = \frac{V(i) - V(j)}{U(i)}$$

where $V(i)$ is the execution commencement time of task i (which is noted in the trace file *traces2*). $U(i)$ is the task's computation time which is just calculated.

6.7.3 Program Databases

In $ATME$, program databases are designed to supply history values of task attributes to the *task model construction* component, which can then predict the task model in the forthcoming execution. $ATME$ employs the following five program databases:

- program database *pd_comp*:

  This database is utilized to store the computation time of parallel tasks in previous executions. In $ATME$, the task computation time is not directly captured at runtime. The calculation of computation time, based on captured node attributes in the task's control flow graph, is presented in Section 6.7.2.

- program database *pd_comm*:

  The magnitude of data transmitted between tasks is stored in this database. Such information is directly captured at runtime by probes and stored in the trace file *traces2*. With the availability of network bandwidth, the communication time between interrelated tasks can be achieved.

- program database *pd_prob*:

  The occurrence of communication between interconnected tasks is retained in the database *pd_prob*. Such information is directly generated by probes, stored in the trace file *traces3*, and used to predict the value of the task attribute, *execution*

probability. Due to conditional branches attached to task runtime operations, the data communication may not take place in certain program executions. *Execution probability*, is defined to capture the likelyhood of such communication.

- program database *pd_psp*:

  This database records the preemption start point of communicating tasks. As stated, in the case where preemption is permitted among parallel tasks, preemption can enhance system performance, as seen in Chapter 4. Furthermore, preemption can also affect the scheduling policy. It is suggested that the scheduling algorithm take this factor into account, as done in *PET*. The task attribute, *preemption start point*, is proposed in this thesis to model preemptive task scheduling and execution. The probe *timing* inserted after a data-transmission operation within a task can assist in obtaining the preemption start point between this parent task and its dependent child task. The calculation of the preemption start point is discussed in Section 6.7.2.

- program database *pd_incp*:

  If the input parameter probe is inserted in the user task (i.e., the user sets the "incps" switch to "on"), then the corresponding trace file, *traces4*, dumps its content to this program database.

Together, the program databases collected by *ATME* provide execution history values of task attributes to other *ATME* functional components. In particular, such profiles are mostly used by the *task model construction* component to predict task attributes in the forthcoming execution. In addition, such profile information is also utilized by the *post-execution analysis* component (Section 6.8) to supply performance reports as well as tuning suggestions to the application programmer.

6.8 Other Components

The remaining components in *ATME* are *post-execution analysis* and *report generation.* These two components are designed to provide tuning suggestions to the application programmer, as well as to *ATME* itself. For instance, tuning suggestions might include "the volume of data communicated between these two tasks is excessive", which may result in network congestion, so that the combination of tasks can be considered. The objective, in general, is to improve program performance.

Tuning suggestions can also be derived from the conditional task model. Equipped with the various program databases, a conditional task model can be determined: directed edges illustrate precedence relationships between parallel tasks, and task attributes describe task behaviour. For example, task communication time correlates to the magnitude of data communicated between interrelated tasks, in particular, execution probability illustrates communication patterns of the parallel program. From the conditional task model, the user can either define constraints to guide the distribution of parallel tasks onto underlying available processors, or adjust task partition by, say, splitting a task into two sub-tasks so as to improve the parallelism.

Tuning suggestions and performance reports may vary, since different application programmers may have different expectations of their parallel programs. At present, the components *post-execution analysis* and *report generation* are mainly adopted to produce experimental results, as presented in Chapter 7. Performance analysis in parallel processing is an active research area which has attracted much attention in recent years. Results can be found in [29, 56, 67, 85, 108, 124, 146, 175]. Promising results can be incorporated into *ATME* at a later date.

Chapter 7

Simulation and Experimentation

This thesis, through the $ATME$ environment, realizes efficient scheduling polices and high system performance for both the conditional task scheduling and the preemptive scheduling problem. In particular, two task scheduling algorithms, namely, CET and PET, are proposed and implemented in $ATME$. This chapter presents experimental results regarding $ATME$ and its adopted algorithms.

Section 7.1 illustrates the mechanisms used to simulate parallel programs, and the underlying parallel and distributed system. Section 7.1 also discusses the simulation of task execution and the framework for the experiments.

Section 7.2 presents experimental results regarding the features of $ATME$ and the conditional task scheduling problem. The experiments illustrate the performance of $ATME$ and compare $ATME$ against both a random scheduling strategy and a round-robin scheduling algorithm. The experiments also reveal the significance of the *execution probability* task attribute in influencing the efficiency of conditional task scheduling. In addition, Section 7.2 examines the responsiveness of $ATME$ to an abrupt change in program usage patterns.

Section 7.3 gives experimental results on preemptive task execution and preemptive task scheduling. It compares the discrepancy in terms of system performance under different preemption strategies. Specifically, a comparison between preemptive and non-preemptive task execution, as well as preemptive task scheduling versus non-preemptive task scheduling, is undertaken.

7.1 Simulation

The simulation of parallel programs, the parallel and distributed system, and the execution of parallel tasks involves the mimicing of a "practical" problem and a "practical" multiprocessor architecture, on which experiments can be undertaken to capture experimental results regarding $ATME$ and related issues. Objects to be modeled in the task scheduling problem include: the target machine (via a processor model), the parallel program (via a task model) and execution of the program (via program usage patterns). Such simulation is presented in Sections 7.1.1, 7.1.2 and 7.1.3, respectively. The simulation results have been validated by spot checking results against actual distributed applications.

Parameters are adopted to simulate each "object". For the sake of clarity, in this section, all simulation parameters are written in *Italics* beginning with a capital letter, while other parameters are shown in **bold** font.

7.1.1 Simulating the Target Machine

The target machine is portrayed by a *processor model*, as introduced in Section 3.1. It is assumed that tasks of a parallel program can execute in parallel on their host processors and communicate with each other through the message-passing mechanism.

The parameter, *ProcNum*, describes the number of available processors in the underlying parallel and distributed system. For any particular processor, the parameter, *LinkNum*, indicates the number of network links from this processor to other processors. The interconnection between a pair of processors in the system is illustrated by the parameter *ProcConProb*, where the value of 0 indicates no connection exists between the two processors while 1 represents a direct connection exists. Consequently, among all possible connections from a particular processor, the number of links (emitting from this processor) on which the value of *ProcConProb* is 1 must be equal to the value of *LinkNum* for this processor.

As stated in Section 3.1, this thesis assumes an "ideal" processor model available. That is to say, all processors are identical and are fully interconnected with identical communication networks. With respect to the simulation of the target machine, the parameter *LinkNum* of any processor is equal to (*ProcNum* - 1) and *ProcConProb* for all links between any two processors is 1 (i.e., connected). Consequently, *ProcNum* uniquely determines a processor model, and thus a target parallel and distributed system, with respect to this thesis.

7.1.2 Simulating the Parallel Program

The simulation of the parallel program is conducted through the description of the *task model* presented in Section 3.2. Such simulation involves a number of aspects: the *task interconnection structure*, *task attributes* and *program usage patterns*. Each aspect is governed by a number of simulation control parameters discussed in Sections 7.1.2.1 through 7.1.2.3.

7.1.2.1 Task Interconnection Structure

The *task interconnection structure* is established to reflect the task precedence relationships in the application program. It is defined by three parameters: *TaskNum*, *TaskSuccNum* and *TasksPLvl*. The *TaskNum* parameter determines the number of tasks in the parallel program. *TaskSuccNum* is the maximum number of child tasks which may be spawned by a task. When constructing (simulating) the task model, this thesis adopts a method in which all tasks of the program are arranged in a hierarchical structure. The parameter *TasksPLvl* describes the maximum number of tasks permitted at each level of the hierarchy. During the simulation, task interconnections are generated randomly between any two levels in the model, to simulate communication and precedence relationships between parallel tasks.

Note that the *task interconnection structure* generated above illustrates all possible message-passing operations between tasks. The occurrence of a task interconnection relationship at runtime depends on the value of the *execution probability* which is associated with it. Therefore, the *task interconnection structure* can be regarded as invariant between different program executions.

7.1.2.2 Task Attributes

The second aspect to be simulated for a parallel program is the *task attributes*, which define the behaviour of the tasks within the program. In this thesis, the task attributes include task computation time, communication time and execution probability, as stated in Section 3.2. The task attribute also includes the preemption start point to illustrate the preemptive task model as presented in Section 4.1.

The values of all attributes of a parallel task are assumed to be determined by an input parameter, denoted as *incp*, for the task. *incp* is not a simulation *control* parameter *per se*, but represents factors which determine the behaviour of the task. As seen in Section 7.1.2.3, the experiments simulates the change of task attributes (between different program executions) through the task's *incp* value, which itself is controlled by a set of simulation control parameters (as seen below). The introduction of *incp* for each parallel task largely reduces the number of simulation control parameters in the experiments, thereby reducing the complexity of the simulation process; otherwise, each task attribute requires a set of simulation control parameters, to manipulate (simulate) the variation of each attribute between program executions. However, simplification by no means results in loss of generality with respect to the experimental results.

The task attributes, in reality, may not be uniquely determined by the task's input parameter, this thesis therefore introduces four simulation parameters, namely, *CompDistb*, *CommDistb*, *ProbDistb* and *PreemptDistb*, to represent the factors, apart from the input parameter *incp*, which affect task behaviour with regard to task computation time, task communication time, execution probability and preemption start point between the tasks, respectively. In the following experiments, all of these parameters are presumed to have value 0, in order to simplify the simulation system and focus on the problems addressed by this thesis.

For a parent task, and its child task, the preemption start point (PSP) is simulated in the range of *PreemptstLB* and *PreemptstUB*. As mentioned in Section 4.1, when $PSP = 1$, it indicates that there is no preemption between the interrelated tasks.

For any generated task model, only the *task attributes* are assumed to vary between program executions, while the number of parallel tasks and the task interconnection structure are regarded as static across the various executions of the parallel program.

The parameter *AvePMRatio* defines the ratio between the average computation time to the communication time of a parallel task. It simulates different kinds of parallel applications: a high value of *AvePMRatio* (e.g., 20.0) models computation-intensive applications, while a low value (e.g., 0.01) represents communication-intensive applications. The simulation covers a variety of different types of parallel programs.

7.1.2.3 Program Usage Pattern

The final aspect in simulating a parallel program is the *program usage pattern.* This is realized by defining the usage pattern of each constituent task as the factor which uniquely determines the behaviour of the task, i.e., the input parameter *incp.* Practically, *incp* can be replaced by a vector or a matrix which describes more than one factor affecting the task behaviour.

For a parallel task in a conditional parallel program, the value of *incp* varies across different program executions, simulating variation in the task's usage pattern. In a program execution, different tasks in the program may have different values of *incp.*

The variation in *incp* for a task is modelled by a periodic function (such as "sine"). Such a variation of *incp* models the changing interaction with, and usage of, the task by human users under normal actual use of the parallel program. The amplitude of the periodic function is defined by two simulation parameters, *RegLB* to *RegUB*, which model the lower and upper bound value of *incp*, respectively. The frequency with which the value of *incp* varies between program executions is determined by the range *FreqLB* to *FreqUB.* Therefore, the variation of a task's usage pattern is portrayed by four simulation control parameters: *RegLB*, *RegUB*, *FreqLB* and *FreqUB.* Both the magnitude and frequency of change in *incp* are defined for each task when generating the task interconnection structure, and may vary from one task to the other. The set

Control Parameter	Values Used in the Experiments
ProcNum	2,3,4,5,6,8,9,10,15,20,25
LinkNum	ProcNum-1
ProcConProb	1.0
TaskNum	5,10,20,30,40,45
TaskSuccNum	min(5,TaskNum/2)
TasksPLvl	TaskNum/2
RegLB, RegUB	5,10
FreqLB, FreqUB	15,20
PreemptstLB, PreemptstUB	20, 100
CompDistb	0.0
CommDistb	0.0
ProbDistb	0.0
PreemptDistb	0.0
AvePMRatio	0.1,0.5,1.0,5.0,10.0
InitRuns	5
Runs	20
Sets	5

Table 7.1. Simulation parameters and their experimental values.

of usage patterns for each task constitutes the entire program's usage pattern.

7.1.3 Simulating Task Execution

The program execution is categorized according to the value of three simulation control parameters: *AvePMRatio*, *TaskNum* and *ProcNum*. Correspondingly, experimental results are analyzed and examined by each category as determined by these three parameters. Within each category, a number of task models and processor models are generated (with different task interconnection structures or task attribute values). The simulation parameter *Sets* represents the number of different task models generated in each category. All experimental results for each category are averaged to collect an unbiased result.

With respect to the simulation of program execution, for any simulated program and processor model (i.e., a specific set of values of *AvePMRatio*, *TaskNum*, *ProcNum*

and a certain parallel program), *InitRuns* number of executions are undertaken to capture initial task runtime data. The same parallel program then executes *Runs* times (with different usage patterns, i.e., with different task attributes) to collect and achieve average performance data.

All simulation parameters and their values used to gather experimental results are listed in Table 7.1.

7.2 Experiments Dealing With Conditional Task Scheduling Issues

This section presents the experimental results with respect to the performance of $ATME$ and issues related to conditional task scheduling (CTS). Section 7.2.1 compares the performance of $ATME$ to the case where the task model is precisely known prior to runtime (which is regarded as the "ideal" situation in CTS). Sections 7.2.2 and 7.2.3 compare the performance of the conditional task scheduling algorithm, CET, against a random scheduling strategy and a round-robin scheduling algorithm, respectively. Section 7.2.4 manifests the significance of the *execution probability* attribute in influencing the efficiency of program execution. Section 7.2.5 shows the responsiveness of $ATME$ in the case when the usage pattern of the parallel program undergoes a sudden change. The accuracy of $ATME$ estimates on task attributes is also given in this section.

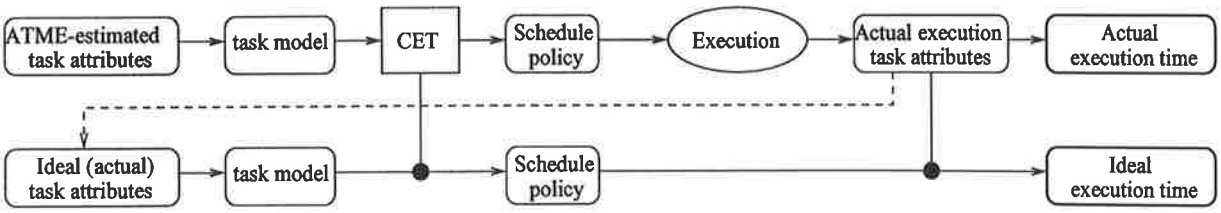


Figure 7.1. The actual and ideal execution time.

7.2.1 System Performance Under *ATME*

The performance of a parallel system employing *ATME* for parallel programming support is measured by the performance difference between *ATME* (i.e., the actual execution time) and the ideal execution time. Figure 7.1 illustrates how the two types of execution time (actual and ideal) are determined.

Prior to program execution, using the predicted task model, a scheduling algorithm, such as *CET*, is used to distribute tasks onto available processors, and thus the *actual execution time* (*ATME Exec. Time*) is obtained.

After program execution, the actual task attributes (which were captured at runtime) are used to construct the "ideal" task model. In other words, task attributes of the "ideal" task model are assigned values equal to what was achieved at runtime. The execution is (virtually) repeated with the ideal task model and the same scheduling algorithm. The the *ideal execution time* is then obtained.

The term, *AIR*, is introduced:

$$AIR = \frac{ATME\ \ Exec.\ \ Time - Ideal\ \ Exec.\ \ Time}{Ideal\ \ Exec.\ \ Time}$$

A negative value of *AIR* indicates that *ATME* outperforms the ideal situation; while a positive *AIR* shows the opposite. The *ideal execution time*, rather than the optimal schedule length, is chosen for comparison against the performance of *ATME*,

AvePMRatio	$AIR < 0$		$AIR = 0$	$AIR > 0$	
	Exec%	AveDiff%	Exec%	Exec%	AveDiff%
0.1	14	7.2	45	41	15.1
0.5	12	3.8	50	38	7.6
1.0	9	2.7	50	41	7.7
5.0	8	3.2	53	39	6.1
10.0	9	3.7	55	36	4.9

Table 7.2. Application program performance by employing $ATME$.

owing to the computational impracticality of determining the optimal scheduling length of a parallel program.

The experimental results are grouped by the three major simulation parameters *AvePMRatio*, $TaskNum$ and $ProcNum$. Table 7.2 shows the difference between the performance of $ATME$ and the ideal, categorized by the *AvePMRatio* parameter. For a fixed value of *AvePMRatio*, executions are divided into three groups, depending on the value of AIR. The first and third groups of AIR values have two sub-columns, representing the percentage of simulated executions falling into this category, and the average difference between $ATME$ execution time and the "ideal execution time", respectively. The second group of AIR values just indicates the percentage of execution in this category, since the average performance difference is 0.

For each category of *AvePMRatio*, it is possible that the performance of $ATME$ is better than that of the ideal (i.e., the $AIR < 0$ column in Table 7.2). This implies that the ideal performance mentioned in this thesis does not necessarily mean the optimal performance of the program. It merely represents the "best" solution anticipated by the scheduling algorithm. Nevertheless, the scheduling algorithm may not produce an optimal distribution solution based on the precise task model.

From Table 7.2, it can be observed that the majority of executions using an inaccurate task model (i.e., predicted by $ATME$ prior to program execution) do not perform as well as the corresponding executions which employ the precise task

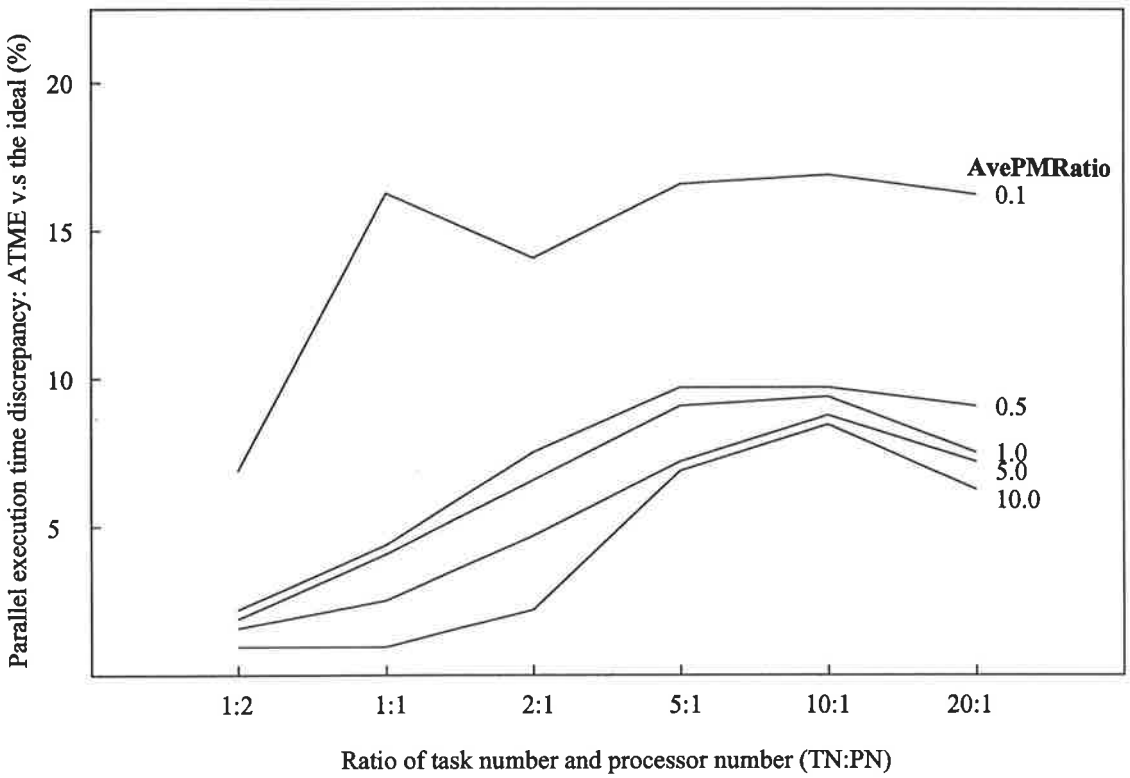


Figure 7.2. $ATME$ performance with various ratios of task number to processor number.

model (i.e., the ideal case). This indicates that the accuracy of the task model, not surprisingly, directly affects the performance of the system. As seen, $ATME$ performs, at worst, around 15% worse than the ideal case when *AvePMRatio* is 0.1 (i.e., communication-intensive applications), while 59% (14% in $AIR < 0$ column and 45% in $AIR = 0$ column) executions under $ATME$ achieve better or the same performance as the corresponding ideal case. In computation-intensive applications (*AvePMRatio* equal to 10.0), the performance difference between $ATME$ and the ideal strategy gets even closer: only 36% of executions under $ATME$ show slightly inferior performance to the ideal case (a mere 4.9% performance discrepancy).

Figure 7.2 shows the difference between the average $ATME$ performance and the ideal performance, under various ratios of task number and processor number. As seen,

for the same type of applications (i.e., when $AvePMRatio$ remains constant), when the number of available processors is greater than the number of parallel tasks, such as $TN : PN = 1 : 2$, the performance discrepancy between $ATME$ and the ideal case is minimal.

There is a significant gap between the results when $AvePMRatio = 0.1$ and the other values of $AvePMRatio$ in Figure 7.2. This indicates that communication-intensive applications are more sensitive to the accuracy of the task model than the other types of applications. Since it is impossible to estimate the task model with 100% accuracy prior to execution, due to the fact that conditional branches and loops within parallel task can not be determined until runtime, this performance difference between a practical environment (such as $ATME$) and the ideal case can not be avoided. Nevertheless, from later experiments in this chapter, it can be observed that $ATME$ performs generally better than the other practical scheduling algorithms.

7.2.2 CET vs. Random Distribution Strategy

A comparison between the performance of CET and that of a random distribution strategy (denoted as $RDIST$) is conducted in this experiment. The $RDIST$ strategy randomly distributes the parallel tasks onto the underlying available processors. It approximates what a naive user (or a user concentrating more on parallel computation algorithms than on the task scheduling issue) might attempt if distributing tasks by hand. For each group of simulation parameters (i.e., $AvePMRatio$, $TaskNum$ and $ProcNum$), a number of task models are simulated to represent different kinds of parallel programs. For each program, CET is used to obtain a scheduling policy (based on previous program executions) and a measure of execution time; and then a random distribution policy is used with the same task model to distribute tasks onto

AvePMRatio	$RAR < 0$		$RAR = 0$	$RAR > 0$	
	Exec%	AveDiff%	Exec%	Exec%	AveDiff%
0.1	2	2	1	98	124
0.5	2	3	1	97	46
1.0	3	4	2	95	35
5.0	4	3	2	94	22
10.0	3	3	2	95	20

Table 7.3. The performance of $ATME$ versus $RDIST$.

processors. The parallel execution time of each program under these two scheduling strategies is compared.

The term RAR is introduced to compare the execution time achieved by the random strategy $RDIST$ against CET:

$$RAR = \frac{RDIST\ Exec.\ Time - CET\ Exec.\ Time}{CET\ Exec.\ Time}$$

A positive value for RAR indicates that CET performs better than the random strategy $RDIST$, while a negative value of RAR shows the opposite.

Table 7.3 shows the results of this experiment. It follows a similar format to Table 7.2. As seen, $ATME$ significantly outperforms $RDIST$ for almost all types of applications — not only in terms of system performance (the AveDiff% column in the rightmost $RAR > 0$ category), but also in terms of the percentage of executions in that category (the Exec% column within the $RAR > 0$ category). For instance, when $AvePMRatio$ is 0.5, that is, the application is slightly communication intensive, 97% of applications show as much as a 46% performance gain when utilizing $ATME$ to schedule tasks.

From Table 7.3, it can also be seen that the performance of applications with an intensive communication load are more sensitive to the scheduling policy than other kinds of parallel applications. A strategy specifically dealing with conditional

communication-intensive applications becomes more crucial. $ATME$ addresses this concern. For example, when $AvePMRatio$ is 0.1, $RDIST$ performs 124% worse than $ATME$ in 98% of executions.

7.2.3 CET vs. Round Robin Algorithm

This experiment aims to compare the efficiency of the CET scheduling algorithm used by $ATME$ and the round-robin task assignment strategy (abbreviated as RRA) which is employed in PVM to distribute parallel tasks onto the virtual parallel machine [69].

For a task model and a processor model generated, at first, the CET scheduling algorithm is used to produce a scheduling policy by which parallel tasks are allocated onto available processors. Therefore, the efficiency of the CET algorithm, measured by the parallel execution time (PT), is obtained after the completion of program execution.

Secondly, with the same task model and processor model used by the CET algorithm as above, the round-robin algorithm, RRA, is employed to undertake task distribution. Following the same manner as above, the parallel execution time of RRA is then determined.

The term RCR is introduced in this thesis to evaluate the efficiency difference between RRA and CET algorithm.

$$RCR = \frac{RRA\ Exec.\ Time - CET\ Exec.\ Time}{CET\ Exec.\ Time}$$

A positive value for RCR indicates that the CET algorithm performs better than the round-robin scheduling strategy. A negative value of RCR shows the opposite.

The experimental results are presented in Table 7.4. According to the value of RCR, the performance comparison between CET and RRA is analyzed into three

AvePMRatio	$RCR < 0$		$RCR = 0$	$RCR > 0$	
	Exec%	AveDiff%	Exec%	Exec%	AveDiff%
0.1	1	2	0	99	161
0.5	6	3	2	93	44
1.0	8	4	2	90	26
5.0	27	4	5	68	8
10.0	32	4	4	64	5

Table 7.4. The performance of CET versus the round-robin scheduling algorithm.

groups. As observed, the CET algorithm generally outperforms RRA in all types of applications. For instance, when $AvePMRatio$ is 0.5, 93% of applications show as much as 44% performance gain when utilizing CET to schedule tasks. The CET algorithm is significantly superior to the round robin algorithm especially in communication-intensive applications. As seen, when $AvePMRatio = 0.1$, 99% executions under RRA performs on average 161% poorer than those under the CET. It can be seen that, for the benefit of system performance, it is important to utilize a scheduling algorithm which is specifically oriented to the conditional parallel programming.

7.2.4 Execution Probability in Conditional Task Scheduling

This section reveals the significance of the task attribute, *execution probability*, in terms of its influence on system performance. In Chapter 3, a conditional task model is presented to describe conditional parallel programs in which message-passing operations may be guarded. The attribute, *execution probability*, is introduced into such a task model and plays a critical role in producing a scheduling policy with the conditional task scheduling algorithm CET.

This experiment compares the efficiency of the scheduling policy when the scheduling algorithm first considers and then ignores the factor of execution probability. These two algorithms are CET (Section 3.6.2) and ERT [115] (which is based on the deterministic task model), respectively.

AvePMRatio	$ECR < 0$		$ECR = 0$	$ECR > 0$	
	Exec%	AveDiff%	Exec%	Exec%	AveDiff%
0.1	10	7.1	53	37	16.1
0.5	5	4.9	49	46	9.9
1.0	3	2.3	46	50	9.5
5.0	3	2.9	47	50	7.5
10.0	3	3.3	48	49	7.1

Table 7.5. Performance of ERT vs. CET.

The experiment is conducted as follows. Prior to program execution, the conditional task model is predicted, based on the previous execution profile. With the anticipated task model and processor model, the algorithm CET is applied so that the performance under the CET policy is determined; this is termed *CET execution time*. The above procedure is replayed with the algorithm ERT employed with the same estimated task model to obtain the *ERT execution time*. ERT treats the conditional task model as a deterministic one, i.e., it does not consider the attribute *execution probability* when determining the scheduling policy, and simply assumes that *execution probability* is 1.0 on all task interconnections.

The comparison between the execution time achieved under CET and ERT is defined below, via the term ECR:

$$ECR = \frac{ERT\ Exec.\ Time - CET\ Exec.\ Time}{CET\ Exec.\ Time}$$

A positive value of ECR indicates that the algorithm CET performs better than ERT, since it indicates that the execution time of the program with ERT-generated scheduling policy is longer than that with the CET policy.

Experimental results are displayed in Table 7.5. When the *AvePMRatio* is 1.0, 50% of the executions show, on average, 9.5% less efficiency under ERT when compared to CET.

In some cases, ERT performs better than CET. When $AvePMRatio$ is 0.1, 10% executions can achieve around 7.1% performance gain when adopting the algorithm ERT to distribute the parallel tasks, as compared to that under CET. However, in the same application category (i.e., the same $AvePMRatio$ value), it can also be observed that CET performs better than ERT in 37% of executions, with the performance gain around 16.1%.

This experiment indicates that the factor of execution probability is important and should be considered by a scheduling algorithm. CET takes this factor into account, and consequently enjoys a performance improvement over ERT.

7.2.5 Responsiveness of $ATME$

When the usage pattern of a parallel program is stable, i.e., the input parameters provided into the program are almost the same over a number of program executions, $ATME$ adaptively achieves an accurate task model, thus gradually approaches the performance of the ideal situation. When the usage pattern varies, $ATME$ needs to collect a new set of execution histories regarding the task model so that it can re-establish a new model in order to reflect the parallel program. This experiment evaluates $ATME$'s performance with respect to how many executions $ATME$ needs in order to achieve ideal performance, after a sudden change in the usage pattern.

The experiment is conducted as follows. For a fixed set of $AvePMRatio$, $TaskNum$ and $ProcNum$ values, $Sets$ number of task models are established, each of which is different from the others in terms of task interconnection structure and task attributes. For each task model, $Runs$ times of program executions are simulated, in which the initial few executions utilize exactly the same task model and the same usage pattern; then the usage pattern of the program is abruptly and sharply changed

AvePMRatio	TN	PN	*Exec*	*AIR*	*AcuComp*	*AcuComm*	*AcuProb*
1.0	45	9	1	0.00	0.90	0.91	1.00
			2	0.00	0.99	0.99	1.00
			3	0.00	1.00	1.00	1.00
			4	0.00	1.00	1.00	1.00
			5	0.00	1.00	1.00	1.00
			6	0.00	1.00	1.00	1.00
			7	0.00	1.00	1.00	1.00
			8	0.00	1.00	1.00	1.00
			9	0.00	1.00	1.00	1.00
			10	0.00	1.00	1.00	1.00
			11	0.94	0.52	0.50	0.34
			12	0.14	0.86	0.87	0.67
			13	0.25	0.63	0.81	0.67
			14	0.00	1.00	1.00	1.00
			15	0.00	1.00	1.00	1.00
			16	0.00	1.00	1.00	1.00
			17	0.00	1.00	1.00	1.00
			18	0.00	1.00	1.00	1.00
			19	0.00	1.00	1.00	1.00
			20	0.00	1.00	1.00	1.00

Table 7.6. The responsiveness of $ATME$.

(simulating a change in task attributes, in particular, the execution probability between interconnected tasks); the new task model is adopted in the remaining of test runs to identify the rate at which $ATME$ tends to the ideal performance.

Table 7.6 displays the experimental results when $AvePMRatio$ is 1.0, and there are 45 parallel tasks (TN column) in the parallel program and 9 processors (PN column) available in the parallel system. 20 ($Exec$ column) test runs takes place for each fixed task model. Half way through (the 11th run), the task model is radically changed and then remains the same for the remaining runs. The AIR column shows the performance discrepancy between $ATME$ and the ideal situation. $AcuComp$, $AcuComm$ and $AcuProb$ represents the accuracy of $ATME$ estimates on task computation time, communication time and execution probability, respectively. When the task model suddenly changes, $ATME$ needs to re-collect the task information and re-build the task model. With only one sharp drop in $ATME$ performance (94%

from the ideal performance), and two cases of performance fluctuation (14% and 25% respectively), *ATME* quickly recovers and adapts to the ideal situation. After three program executions from the pattern change, *ATME* achieves ideal performance, and the accuracy of *ATME* estimates of task attributes tend to 100% also.

Small continuous variations to the usage patterns result in small incremental changes to the task model and task attributes as *ATME* continues to strive for higher performance. The responsiveness of *ATME* depends on the fluctuation of the usage patterns of the parallel program and the amount of execution history retained by *ATME* in order to predict and construct the task model. The more stable the usage pattern, the quicker *ATME* adapts. When the pattern is reasonably stable, the shorter the execution history required, the faster the adaption is. In the experiment, 3 past executions are utilized to predict the task model in the future execution. The user of the parallel program should know how to use the program (i.e., sharp or gradual usage pattern change between executions), therefore, the number of execution histories adopted in the task model construction is left to the user to determine. If sharp changes in usage patterns are expected, then the execution history should be fairly short; however, if gradual changes are expected, then a more detailed execution history may be maintained.

7.3 Experiments Dealing With Preemptive Task Execution and Scheduling Issues

Experiments conducted in this section show the performance improvement of preemptive task execution and preemptive task scheduling. In this section, it is assumed that task runtime operations are not associated with any conditions or loops.

In preemptive task execution, the $N\alpha P * \beta P$ strategy is applied to task execution, i.e., data transmission from a task is permitted to take place before the task completes its execution, while task execution on the same processor is non-preemptive. A large number of parallel programs are simulated by simply extracting their task attributes regarding task computation time, intertask communication time and preemption start point. The experimentation is undertaken in the same manner as previous sections. Parallel applications are classified according to *AvePMRatio*. A range of experiments with varying values of *AvePMRatio* are undertaken with different task numbers, different task interconnections and various numbers of processors to obtain the average performance for each situation.

Three kinds of experimental results for each simulated parallel program are studied and compared. The first type of results are related to system performance in the non-preemptive program execution. That is, ignore the preemption in all constituent tasks, undertake the scheduling procedure (by employing the ERT algorithm [115]) to distribute tasks onto processors. The "non-preemptive" performance is therefore obtained.

The second type of experimental results deals with preemptive task execution (PTE). That is, when the tasks are allocated (prior to execution), ignore the existence of task preemption. At runtime, the strategy $N\alpha P * \beta P$ is adopted to direct program execution. Therefore, system performance using PTE is determined.

Finally, system performance of preemptive task scheduling (PTS) is examined. The scheduling algorithm PET (provided with the preemptive task model) is used to generate a scheduling policy to distribute tasks. Thus, "preemptive task scheduling" performance results can be captured.

The term *GNR* is introduced to measure the parallel execution time difference

AvePMRatio	$GNR > 0$		$GNR = 0$	$GNR < 0$	
	Exec%	AveDiff%	Exec%	Exec%	AveDiff%
0.1	0	0	2	98	4
0.5	0	0	1	99	10
1.0	0	0	2	98	10
5.0	0	0	9	91	7
10.0	0	0	9	91	7

Table 7.7. Performance comparison between preemptive task execution and non-preemptive execution.

AvePMRatio	$PNR > 0$		$PNR = 0$	$PNR < 0$	
	Exec%	AveDiff%	Exec%	Exec%	AveDiff%
0.1	5	10	4	91	6
0.5	1	1	1	98	12
1.0	3	1	1	96	14
5.0	2	1	2	96	17
10.0	1	1	1	98	20

Table 7.8. Performance comparison between preemptive task scheduling and non-preemptive scheduling.

between preemptive task execution (PTE) and non-preemptive execution ($NPTE$):

$$GNR = \frac{PTE\ Exec.\ Time - NPTE\ Exec.\ Time}{NPTE\ Exec.\ Time}$$

A positive value for GNR indicates that PTE is inferior to $NPTE$; while a negative GNR shows the opposite. In the same way, PNR is defined as the performance discrepancy between preemptive task scheduling (PTS) and the non-preemptive case:

$$PNR = \frac{PTS\ Exec.\ Time - NPTS\ Exec.\ Time}{NPTS\ Exec.\ Time}$$

Tables 7.7 and 7.8 list experimental results of the performance comparison between PTE and $NPTE$, as well as between PTS and $NPTS$, respectively. Each table contains experimental data of three groups, depending on the value of GNR and

PNR accordingly. For each group and value of *AvePMRatio*, the table displays the percentage of applications (Exec%) falling in this group and the average value of *GNR* or *PNR* (AveDiff%), respectively. In the middle group of both Tables 7.7 and 7.8, the AveDiff% value for both GNR and PNR is 0, therefore, it is not listed in the table. The context switch time between tasks on the same host is ignored in the experiments.

From Table 7.7, in over 90% of all simulated parallel applications, preemptive task execution shows better performance than non-preemptive execution. In all executions, PTE performs better than, or equal to, the $NPTE$ strategy. This is also theoretically proved in Section 4.3.6.

As observed in Table 7.8, when *AvePMRatio* is 10.0, preemptive scheduling can achieve up to 20% better performance than non-preemptive scheduling in 98% executions.

In a very limited number of cases, *PTS* performs worse than *NPTS*, due to the non-deterministic characteristics of the task scheduling problem. For instance, when $AvePMRatio = 0.1$ (communication-intensive applications), though PTS can be inferior to $NPTS$ by as much as 10%, note that only 5% of all executions fall in this category.

In general, both the theoretical discussion and extensive experiments indicate that consideration of preemption in task execution and scheduling can improve system performance quite dramatically.

Chapter 8

Conclusions and Future Work

As stated in Chapter 1, two objectives of this thesis have been identified: the performance enhancement of parallel and distributed systems, and the support for application programmers in parallel program development. This thesis realizes the first objective through the proposed task scheduling strategies, and the second objective is implemented through the *ATME* environment.

Section 8.1 summarizes the study conducted in this thesis. It also proposes directions for the future work in Section 8.2.

8.1 Conclusions

This thesis studies three major problems: conditional task scheduling, preemptive task execution and task scheduling, and conditional parallel programming support. The thesis presents an environment, named *ATME*, which practically implements the theoretical research regarding these three problems and efficiently serves parallel application programmers.

As reviewed in Chapter 2, task scheduling has been conjectured to be a critical step in parallel processing, since it significantly influences the performance of the parallel and distributed system. The task scheduling problem has been studied for quite some time, and can be traced back to as early as 1960s. It has been theoretically proved that this problem has no optimal solution within polynominal computation time [168]. A compromise must be made between the performance improvement brought by an efficient scheduling policy and the cost of achieving such a scheduling solution. Many heuristics strive to obtain a generally "good" scheduling policy within acceptable complexity. To date, most heuristics are based on *a priori* knowledge of the task model of the program being precisely available when the scheduling policy is generated, on the assumption that the task model does not vary between program executions (such as the case in deterministic task scheduling). Furthermore, current research generally focuses on non-preemptive task scheduling in which a child task can not commence its execution until all its parent tasks complete. Extra effort is required to develop task scheduling research so that general application problems can be efficiently dealt with.

It is stated in Chapter 2 that the task scheduling process in parallel program development should be automated, in order to relieve the burden on the application programmer who can therefore concentrate on program design [52]. In addition, with the vast effort put into scheduling algorithms, such automation can make use of the most efficient algorithm available and thus achieve high system performance.

Under these circumstances, Chapter 3 studies the scheduling problem within conditional parallel programming, i.e., conditional task scheduling. Conditional branches may be associated with task runtime operations (in particular, task spawn, data transmission and data reception). Deterministic task scheduling (which is handled by most of current scheduling research) is a special case of conditional scheduling. In

conditional scheduling, the task model, which reflects task runtime operations, can not be precisely known prior to program execution, since values of the associated conditional branches can not be determined until runtime. Furthermore, a good scheduling policy for an individual execution does not guarantee the same efficiency for other executions. At present, little effort has been made in conditional task scheduling.

Chapter 3 proposes a conditional task model to reflect the behaviour of parallel tasks which may be conditionally spawned and communicate between each other. In particular, a new task attribute, *execution probability*, is introduced to describe the conditional execution between tasks. On the whole, the strategy to tackle the conditional task scheduling problem is composed of two aspects: the construction of the task model and a conditional task scheduling algorithm named CET which is applied to the conditional task model. Task model construction is based on execution profiles of the program, which are captured at runtime and retained. Two techniques are employed in the task model construction. They are: linear regression model and the finite state machine.

This thesis also examines the problem of preemptive task execution and preemptive task scheduling in Chapter 4. Preemption between parallel tasks at runtime indicates that data communication is permitted to occur at any point within tasks, rather than being restricted to the task's beginning (in the case of data reception) and end (in the case of data transmission). Chapter 4 studies in detail preemptive task execution and its influence on system performance. It is demonstrated that preemptive execution does not always result in system performance improvement. A strategy is proposed to guarantee the performance enhancement in preemptive execution.

Another issue studied in Chapter 4 is preemptive task scheduling. A preemptive task model is introduced to describe preemption between tasks. Chapter 4 presents a

scheduling algorithm, PET, to deal with preemptive task scheduling. The construction of the preemptive task model follows the same strategy as that of the conditional task model, as presented in Section 3.5.

Chapter 5 in this thesis discusses programming support for the development of (conditional) parallel programs, which is widely accepted as intrinsically tedious and complex for an application programmer to manually deal with. Programming support is established over PVM, with the focus on operations of conditional task spawn, conditional data transmission and conditional data reception. The support is demonstrated through a set of $ATME$ runtime primitives. It is shown that the work-load on the programmer is significantly reduced, through the assistance of $ATME$ primitives. The programmer can therefore focus on program design, rather than be distracted by operational issues raised in conditional parallel programming. Furthermore, the $ATME$ task spawn primitive dynamically accesses the scheduling policy file generated by $ATME$. In addition, $ATME$ realizes the automation of the task scheduling process, so that the programmer is freed of the need to consider this complex issue in parallel processing.

On the basis of the research work undertaken, this thesis presents an environment, named $ATME$. Chapter 6 illustrates the $ATME$ environment and elaborates each $ATME$ functional component in detail. Taking as input a user-developed parallel program (with $ATME$ runtime primitives), $ATME$ can then be invoked to analyze, instrument and preprocess the user-provided program, so that the resultant program is ready to run on the PVM platform and gather runtime information with respect to parallel tasks. $ATME$ also establishes a processor model to reflect the underlying parallel and distributed system. Through a few initial test runs, $ATME$ accumulates knowledge, possibly incomplete, of attributes and precedence relationships of parallel

tasks, and then establishes the task model for the next program execution. The *ATME* self-contained scheduling algorithm (a combination of *CET* and *PET*) is triggered with the inputs of the task model and the processor model, and generates a scheduling policy which arranges the allocation of parallel tasks onto the available processors, and manages the execution order of tasks assigned on the same host. Probes inserted into user tasks capture task runtime information which is retained in program databases for the construction of the task model in later program executions. As the program is continuously invoked and executed, the adaptive task model is accurate enough to reflect the configuration and behaviour of the application problem, i.e., precedence relationships illustrated by the task model can describe the configuration of the program, and the value of task attributes mirrors the behaviour and communication among parallel tasks. Therefore, the scheduling algorithm, provided with an increasingly accurate task model, can produce an efficient scheduling policy.

The environment is ready to assist in the development of conditional parallel programs. It imposes no extra burden on the user of *ATME*. Each functional component within *ATME* has designed with a clear interface, and can be easily replaced as future advances occur. Therefore, new research work can be efficiently incorporated into *ATME* in order to further improve system performance.

Chapter 7 presents experimental results to manifest the properties and efficiency of the strategy presented in this thesis to deal with the conditional task scheduling as well as preemptive task execution and scheduling problems. Experiments are undertaken in simulating various parallel applications and systems, for instance, from computation-intensive applications to communication-intensive applications. The performance of *ATME* is compared to the "ideal" system performance achieved when it is assumed that the task model is precisely known prior to execution. *ATME* is shown to achieve

high system performance when usage patterns of the parallel program are relatively stable, owing to the fact that the task model can be estimated with high accuracy, as compared to the actual task model at runtime. $ATME$ is also experimentally compared to a random scheduling strategy ($RDIST$). The results indicate that $ATME$ is largely superior to $RDIST$. An comparison between the CET and the round-robin scheduling algorithms is undertaken, and the results verify that CET can achieve much higher system performance than the round-robin algorithm used by PVM in communication-intensive applications.

Experiments show that the newly-introduced task attribute, namely *execution probability*, is critical to the efficiency of the scheduling policy, and to system performance. It can be concluded that a scheduling algorithm which considers this attribute, such as CET, generally performs better than other options do. The responsiveness of $ATME$ to radical change in program usage patterns is tested. $ATME$ establishes the task model in the forthcoming program execution, on the basis of a number of previous executions. This ensures that $ATME$ responds to evolving usage patterns but not at such a rapid rate that a single execution which does not fit the usual user profile forces a radical change in the scheduling policy. $ATME$ demonstrates a quick "recovery" in the case of abrupt change in usage patterns.

Section 7.3 provides experimental results with respect to the performance achievement of preemptive task execution and preemptive task scheduling, respectively. It is shown that the $N\alpha P * \beta P$ strategy directing preemptive task execution can result in performance enhancement. Furthermore, considering preemption between tasks can alter the scheduling policy which is obtained by non-preemptive scheduling algorithms, thus improve system performance further.

8.2 Future Work

The work discussed in this thesis has a number of directions which may be pursued as future research.

The scheduling algorithms proposed in this thesis, i.e., CET and PET for conditional task scheduling and preemptive task scheduling, respectively, can be extended, in order to pursue even better system performance than currently achieved. One possible direction is to introduce more factors to be considered by the algorithm. Intuitively, the more factors taken into consideration by a scheduling algorithm, the more efficient the generated scheduling policy may be. For instance, in this thesis, two attributes, *execution probability* and *preemption start point*, are introduced into the task model to illustrate the conditional and preemptive parallel program. Experimental results presented in Chapter 7 demonstrate benefits (in terms of system performance) brought about such consideration.

As seen, this thesis adopts an "ideal" parallel and distributed system model, in which all constituent processors are regarded as identical and fully-connected by identical communication networks. In general, processor heterogeneity should be considered and reflected by the scheduling algorithm. Further, more attributes are required by the processor model than proposed in Section 3.1, in order to adequately describe such a system. The design of the $ATME$ environment has provided interfaces for such extension.

A task model utilizing two task attributes, i.e., computation time and communication time, has been widely adopted in current scheduling research. This thesis introduces two more attributes, i.e., execution probability and preemption start point, to describe parallel task behaviour. Experiments have indicated that

the conditional (and preemptive) task model presented can achieve better system performance than a model which merely describes task computation time and communication time. However, it has not yet been proved that such task attributes are sufficient to authentically reflect the requirements of a parallel program. It is possible that other attributes must be introduced into the task model to obtain a more accurate reflection of the parallel tasks and their relationships.

This thesis employs two techniques, the linear regression model and the finite state machine, to predict task attributes in the forthcoming execution, on the basis of corresponding values in past execution profile. Alternative techniques, such as fuzzy logic and statistical analysis, may be utilized to replace the current ones, in order to estimate the task model with even greater accuracy than what can be achieved at present.

The dynamic approach can be adopted to tackle the conditional task scheduling problem. In this case, the estimation of task attributes can be expected to be more accurate than in the static approach. Although runtime overhead will be incurred, the dynamic approach is still a valuable direction to achieve system performance in parallel processing.

The discussion of the conditional parallel programming support presented in this thesis concentrates on three major primitives: conditional task spawn (*tme_spawn*()), conditional data transmission (*tme_send*()) and conditional data reception (*tme_recv*()). More primitives can be incorporated into the *ATME* runtime library to better assist the program development. In particular, the distinction of multiple task instances (multiple processes corresponding to exactly the same source code) requires more attention. This problem becomes prominent when managing the point-to-point data communication in which the unique identification of the

communicating task is necessary. At present, this problem is basically left to the application programmer to deal with. Additional effort can be made to relieve the programmer of this tedious work.

A graphical interface can be built to present the application programmer with improved conditional programming support. The recognition of multiple task instances (as mentioned above) can make use of such a graphical interface to describe and automatically establish task configuration of the parallel program. Code generation can also be undertaken, with the information provided through the graphical interface, as done in *VPE* and *HeNCE*.

The current control flow graph used by *ATME* to describe the program flow inside a parallel task merely deals with typical constructs in a programming language, i.e., assignment, conditional branch and repetitive loop. Additional mechanisms can be introduced to illustrate other constructs in a language, so that *ATME* becomes more practical than that at present. As stated in Chapter 6, the inserted probes, for the purpose of runtime capture of task information, can alter a task's behaviour to some extent. Techniques for probe impact elimination should be examined, over and above those proposed and realized in this thesis.

Two *ATME* components, *post-execution analysis* and *report generation*, are basically adopted for experimental purposes in this thesis. They can do much more in terms of providing tuning suggestions to the application programmer and even to *ATME* itself. Performance analysis is an area which attracts significant research attention recently, with the aim of developing high performance parallel programs.

ATME is deliberately designed to accept new and advanced techniques without delay. All future work discussed above, once realized, can be immediately incorporated into the *ATME* environment, so that the user is more efficiently and quickly assisted

in the development of parallel programs.

Appendix A

ATME Execution Monitor

This chapter presents the design and implementation of the execution monitor (*EM*) of the *ATME* environment. Section A.1 introduces the data structures adopted in *EM*. The full code is listed in Section A.2.

A.1 Data Structures in *EM*

```
/*-----------------------------------------------------------------*
  Incoming message structure: event (request) from the user task
                              to the EM.
 *-----------------------------------------------------------------*/
struct  in_msg   {
  int           tidxno;         // Event sender's task index number.
  int           tidno;          // Event sender's task id number,
                                // if any.
  int           event_type;     // Event type, determining which
                                // structure below to be used.
  union   {
    int         partidxno;      // tidxno of the parent task.
    int         tidno;          // tidno of the sending task.
    int         chdtidxno;      // tidxno of the child task.
    int         procidxno;      // Processor id.
  } u1;
  union   {
    int         chdtidxno;
    int         partidxno;
    int         tidno;
    int         execseqno;      // Execution sequence of the task.
  } u2;
```

```
};


/*-------------------------------------------------------------*
  Outgoing message structure: event (response) from the EM to the
                              user task.
 *-------------------------------------------------------------*/
struct  out_msg   {
  union  {
    int         tidno;          // Task id number.
    int         execready;      // Mark of "ready to run".
  }  u1;
  union  {
    int         comm_mark;      // Mark of "communication occurs".
  }  u2;
};


/*-------------------------------------------------------------*
  List "tidno": retain spawning status of user tasks, to provide
                the "spawn" or "not-spawn" response.
 *-------------------------------------------------------------*/
struct  list_tidno   {
  int           rec_mark;       // Mark for the record in the list.
  int           tidxno;
  int           tidno;
  int           curparnum;      // Current number of parent tasks.
  int           execready;      // Whether it is its turn to run.
}
struct  list_tidno              ltidno[MAXTASKNUM];


/*-------------------------------------------------------------*
  List "commmark": retain the communication status of a pair of
                   interrelated tasks.
 *-------------------------------------------------------------*/
struct  list_commmark   {
  int           partidxno;
  int           chdtidxno;
  int           comm_mark;
};
struct  list_commmark           lcommmark[MAXTASKNUM * MAXSUCCNUM];
```

```
/*-------------------------------------------------------------*
  List "texec": retain the execution status of user tasks, for the
                analysis of "execution probability" between tasks.
 *-------------------------------------------------------------*/
struct  list_texec   {
  int           partidxno;
  int           chdtidxno;
};
struct  list_texec              ltexec[MAXTASKNUM * MAXSUCCNUM];


/*-------------------------------------------------------------*
  Waiting queue "tidno": hold the event which is waiting for the
                         task identification number of a user task.
 *-------------------------------------------------------------*/
struct  wqueue_tidno   {
  int                   used_mark;
  struct in_msg         inmsg;      // Hold off the entire message.
};
struct  wqueue_tidno            wqtidno[MAXTASKNUM * MAXSUCCNUM];


/*-------------------------------------------------------------*
  Waiting queue "commmark": hold the event which inquiries the
                            communication status between a parent
                            and a child tasks.
 *-------------------------------------------------------------*/
struct  wqueue_commmark   {
  int                   used_mark;
  struct in_msg         inmsg;
};
struct  wqueue_commmark         wqcommmark[MAXTASKNUM * MAXSUCCNUM];


/*-------------------------------------------------------------*
  Waiting queue "execready": hold the event which asks for whether
                             a task is ready to execute or not on
                             its resident processor.
 *-------------------------------------------------------------*/
struct  wqueue_execready   {
  int                   used_mark;
  struct in_msg         inmsg;
};
struct  wqueue_execready        wqexecready[MAXTASKNUM];
```

```
/*-------------------------------------------------------------*
  Pointers to all lists and waiting queues used by the EM task
 *-------------------------------------------------------------*/
struct  pointers   {
  int           ltidno;
  int           lcommmark;
  int           ltexec;
  int           wqtidno;
  int           wqcommmark;
  int           wqexecready;
};
struct  pointers                ptr;
```

A.2 Implementation of *EM*

This section presents the implementation of the execution monitor (*EM*), which is the core component of the conditional parallel programming support in *ATME*. The interaction between the execution monitor and the user task is studied in Chapter 5.

```
/*-------------------------------------------------------------*
  Execution monitor process: recording down the execution path of
  the dynamic task execution of the application; providing the
  inquiries from user processes regarding whether a task has spawned
  or sent data to another one.
  Receiving the requests from user processes, and returning the
  results.
 *-------------------------------------------------------------*/
#include                <stdio.h>
#include                <stdlib.h>
#include                <string.h>
#include                <errno.h>
#include                "pvm3.h"

#include                "atme_0ext.h"
#include                "atme_0global.h"
#include                "atme_0es_global.h"


main(int argc, char **argv)
{
  int f_end, f_tidno;
  struct in_msg inmsg;
```

```
struct out_msg outmsg;
int   f_count = 0;
int   i;


initialization();
f_tidno = pvm_mytid();

f_end = 0;
while  (f_end == 0)   {
  inmsg = get_mq1();

  switch  (inmsg.event_type)   {
    case  EV1_DETCHDSPAWN:
      evproc_detchdspawn(inmsg);
      break;
    case  EV1_SPAWN:
      evproc_spawn(inmsg);
      break;
    case  EV1_TEXEC:
      evproc_texec(inmsg);
      break;
    case  EV1_NOSPAWN:
      evproc_nospawn(inmsg);
      break;
    case  EV1_SEND:
      evproc_send(inmsg);
      break;
    case  EV1_NOSEND:
      evproc_nosend(inmsg);
      break;
    case  EV1_DETPARSEND:
      evproc_detparsend(inmsg);
      break;
    case  EV1_DETEXECREADY:
      evproc_detexecready(inmsg);
      break;
    case  EV1_NEXTEXEC:
      evproc_nextexec(inmsg);
      break;
    case  EV1_EXIT:
      f_end = 1;
      break;
  }
```

```
  }

  dumptexec();
  pvm_exit();
  exit(1);
}



/*-------------------------------------------------------------*
                          Initialization
 *-------------------------------------------------------------*/
initialization()
{
  int  i, j, f_taskidxno;

  loadtasks();
  loadpolicy();

  for  (i=0; i<MAXTASKNUM; i++)   {
    ltidno[i].rec_mark = NOT_PROCESSED;
    ltidno[i].tidxno = tasks[i].taskidxno;
    ltidno[i].tidno = 0;
    ltidno[i].curparnum = tasks[i].parnum;
    ltidno[i].execready = NOT_EXECREADY;
    if  (tasks[i].chdnum == 0)   break;
  }
  ptr.ltidno = i+1;

  for  (i=0; policy[i].procidxno!= MAXNUM; i++)   {
    if  (policy[i].execseqno != 0)   continue;
    f_taskidxno = policy[i].taskidxno;
    for  (j=0; j<ptr.ltidno; j++)
      if  (f_taskidxno == ltidno[j].tidxno)   break;
    ltidno[j].execready = EXECREADY;
  }

  for  (i=0; i<MAXTASKNUM * MAXSUCCNUM; i++)   {
    lcommmark[i].partidxno = 0;
    lcommmark[i].chdtidxno = 0;
    lcommmark[i].comm_mark = 0;
  }

  for  (i=0; i<MAXTASKNUM * MAXSUCCNUM; i++)   {
```

```
    ltexec[i].partidxno = MAXNUM;
    ltexec[i].chdtidxno = MAXNUM;
  }

  for  (i=0; i<MAXTASKNUM; i++)   wqtidno[i].used_mark = 0;
  for  (i=0; i<MAXTASKNUM * MAXSUCCNUM; i++)
    wqcommmark[i].used_mark = 0;
  for  (i=0; i<MAXTASKNUM; i++)   wqexecready[i].used_mark = 0;

  ptr.lcommmark = 0;  ptr.wqtidno = 0;  ptr.wqcommmark = 0;
  ptr.wqexecready = 0;  ptr.ltexec = 0;
}



/*********************************************************************
                          Event Processing
 *********************************************************************/

/*-------------------------------------------------------------------*
  Event Processing: ev1_detchdspawn:
  - This event comes in when the user process tries to detect whether
    the task (indicated by 'taskidxno') has been spawned or not.
  - If the record is locked, i.e., this task is currently spawned by
    some other task, its 'taskidxno' will be put into the list
    'spawn' in a short time.  So, put the event into waiting queue
    'wq_tidno'.  Waiting for other events to wake it up.
  - If the record is found, i.e., this task has been spawned by some
    other task, then return its 'taskidxno'.
  - If there is no exact record, check list 'not_spawn', if still
    can not find it, i.e., this task has not been spawned yet, put
    a new record into list 'spawn' with the 'rec_mark' field set
    to 'locked'.
 *-------------------------------------------------------------------*/
evproc_detchdspawn(struct in_msg  inmsg)
{
  int i, f_tid;
  struct  out_msg   f_outmsg;

  f_outmsg.u2.comm_mark = 0;

  f_tid = get_list_tidno(inmsg.u2.chdtidxno);

  switch  (f_tid)   {
```

```
    case  NOT_PROCESSED:
      mod_list_tidno(LOCKED, inmsg.u2.chdtidxno, TO_BE_AVAIL);
      f_outmsg.u1.tidno = NOT_SPAWN;
      put_mq0(inmsg.tidno, f_outmsg);
      break;

    case  LOCKED:
      put_wqueue_tidno(inmsg);
      break;

    default:
      f_outmsg.u1.tidno = f_tid;
      put_mq0(inmsg.tidno, f_outmsg);
      break;
  }
}



/*-----------------------------------------------------------------*
  Event Processig: ev_spawn:
  - this event occurs when a task has been spawned by one of its
    parent tasks.
  - record the 'tidno' of the newly-spawned task into list
    'spawn'.
  - if there are tasks in the waiting queue 'tidno', and waiting
    for the 'tidno' of the newly-spawned task, wake it up, and
    return 'tidno'.
  - Put a record into list 'texec': indicating the occurrence of
    communication between parent and child task, for the purpose of
    collecting runtime information regarding execution probability
    for use by the scheduling algorithm CET.
*-----------------------------------------------------------------*/
evproc_spawn(struct in_msg  inmsg)
{
  int  i;
  struct  out_msg  outmsg;

/*** PUT INTO LIST tidno && CHECK WAITING QUEUE wqtidno ***/
  mod_list_tidno(SPAWNED, inmsg.u1.chdtidxno, inmsg.u2.tidno);

/*** PUT INTO LIST texec, INDICATING THE HAPPENING OF EXECUTION
     BETWEEN TWO TASKS --- FOR THE USE OF CALCULATING EXECUTION
     PROBABILITY ***/
```

```
  put_list_texec(inmsg.tidxno, inmsg.u1.chdtidxno);
}


/*-----------------------------------------------------------------*
  Event Processing: ev1_nospawn:
  - Substract 1 from the child task in list 'parnum', representing
    the number of parents which may spawn this child task has been
    reduced by 1.  When the 'parnum' of this child task becomes 0,
    i.e., there are no tasks that spawn this child task, the 'tidxno'
    of this child task will then be put into the list 'notspawn'.
  - If the not_spawned task is the parent task for other child tasks,
    then the grand-child task will be affected as well.
  - If some task is not spawned by any of its parent tasks, its
    execution sequence should be skipped and set the next task after
    this task to be 'execready'.
 *-----------------------------------------------------------------*/
evproc_nospawn(struct in_msg  inmsg)
{
  int   i, f_taskptr, f_taskptr2, f_chdnum, f_found, f_taskidxno;


/*** SUBTRACT 1 FROM TASK REFERED BY inmsg.u2.chdtidxno IN LIST
     parnum ***/
  i = 0;
  while  ((i < ptr.ltidno) &&
          (ltidno[i].tidxno != inmsg.u2.chdtidxno))   i++;
  ltidno[i].curparnum--;
  f_taskptr = i;


/*** PUT A RECORD IN LIST commmark && CHECK WAITING QUEUE
     comm_mark ***/
  put_list_commmark(inmsg.u1.partidxno,inmsg.u2.chdtidxno,NO_COMM);


/*** MODIFY LIST tidno ***/
  if  (ltidno[f_taskptr].curparnum == 0)   {
    if  (ltidno[f_taskptr].execready == EXECREADY)   {
      f_taskidxno = nexttask_torun(ltidno[f_taskptr].tidxno);
      if  (f_taskidxno != NO_NEXTTASK)
        mod_list_tidno2(f_taskidxno, EXECREADY);
    }
```

```
    f_taskptr2 = inmsg.u2.chdtidxno;
    f_chdnum = tasks[f_taskptr2].chdnum;
    for  (i=0; i<f_chdnum; i++)   {
      inmsg.u1.partidxno = f_taskptr2;
      inmsg.u2.chdtidxno = tasks[f_taskptr2].chds[i].taskidxno;
      /* other fields in "inmsg" are of no use */
      evproc_nospawn(inmsg);
    }
  }
}



/*-----------------------------------------------------------------*
  Event Processing: ev1_texec
  - Put into list 'texec' the occurrence of execution between
    two tasks: the spawn attempt (intending to spawn a child task
    which has already been spawned by one of its other parent tasks).
 *-----------------------------------------------------------------*/
evproc_texec(struct in_msg  inmsg)
{
  put_list_texec(inmsg.u1.partidxno, inmsg.u2.chdtidxno);
}



/*-----------------------------------------------------------------*
  Event Processing: ev1_send
  - Put a new record into list 'commmark'
  - Check waiting queue 'wqcommark'
 *-----------------------------------------------------------------*/
evproc_send(struct in_msg  inmsg)
{
  put_list_commmark(inmsg.u1.partidxno, inmsg.u2.chdtidxno, COMM);
}



/*-----------------------------------------------------------------*
  Event Processing:
  - Put a new record into list 'commmark'
  - Check waiting queue 'wqcommmark'
 *-----------------------------------------------------------------*/
```

```
evproc_nosend(struct in_msg  inmsg)
{
  put_list_commmark(inmsg.u1.partidxno, inmsg.u2.chdtidxno, NO_COMM);
}



/*-----------------------------------------------------------------*
  Event Processing: ev_detparsend
  - Detecting whether a parent task 'partidxno' has sent data or
    will not send data or may send data to the child task
    'chdtidxno'.
  - Check list 'commmark' to see whether the parent task has sent
    data to child task.
  - If found, issue the sent response to the user process.
  - If not found, put into the waiting queue 'wqcommmark'.
 *-----------------------------------------------------------------*/
evproc_detparsend(struct in_msg  inmsg)
{
  int  f_commmark;
  struct  out_msg f_outmsg;


  f_commmark = get_list_commmark(inmsg.u2.partidxno,
                                 inmsg.u1.chdtidxno);
  switch  (f_commmark)   {
    case  NOT_FOUND:
      put_wqueue_commmark(inmsg);
      break;

    default:
      f_outmsg.u1.tidno = get_list_tidno(inmsg.u2.partidxno);
      f_outmsg.u2.comm_mark = f_commmark;
      put_mq0(inmsg.tidno, f_outmsg);
      break;
  }
}



/*-----------------------------------------------------------------*
  Event Processing: ev_detexecready
  - Detect whether the current task (indicated by 'tidxno' in
    'inmsg') is ready to run, in terms of execution sequence which
```

```
    is determined by the scheduling policy.
  - Check list 'tidno' to see whether this current task is set on
    'execready'.
  - If yes, return, using the field 'tidno' in 'outmsg'.
  - If no, put it into the waiting queue 'wqexecready', waiting for
    another event 'nextexec' to activate it.
 *-------------------------------------------------------------------*/
evproc_detexecready(struct in_msg  inmsg)
{
  struct list_tidno f_ltidno;
  struct  out_msg f_outmsg;


  f_ltidno = get_list_tidno2(inmsg.tidxno);
  switch   (f_ltidno.execready)   {
    case  EXECREADY:
      f_outmsg.u1.execready = 1;
      f_outmsg.u2.comm_mark = f_nothing;
      put_mq0(inmsg.tidno, f_outmsg);
      break;

    case  NOT_EXECREADY:
      mod_list_tidno2(inmsg.tidxno, EXECREADY_WAIT);
      put_wqueue_execready(inmsg);
      break;
    default:
      break;
  }
}



/*-------------------------------------------------------------------*
  Event Processing: ev_nextexec
  - Accept a message which says the current task has finished, the
    next task which is assigned on the same processor as the
    current task and immediately following the current task can go.
    The execution sequence is decided by the scheduling policy
    generated by the scheduling algorithm.
  - Check 'policy' to get the next-go task.
  - Check whether 'execready' in list 'tidno' is execready_wait: If
    it is, activate the corresponding record in 'wqexecready'; if
    not, go on to the next step.
  - Set field 'execready' in list 'tidno'.
```

```
  - Each task (uniquely identified by 'tidxno') has only one
    possible task which is just in front of this task in the
    'gantt' chart.  Therefore, the 'execready' of the next task must
    be either execready_wait or not_execready.  It cannot be
    execready.
 *-------------------------------------------------------------------*/
evproc_nextexec(struct in_msg  inmsg)
{
  int  i = 0, j = 0, f_taskidxno;
  struct  out_msg f_outmsg;
  struct list_tidno f_ltidno;


/* THIS ACTION IS NOT NECESSARY, HERE JUST KEEP ALL DATA FIELDS
   "CLEAN" */
  mod_list_tidno2(inmsg.tidxno, EXEC_END);

  f_taskidxno = nexttask_torun(inmsg.tidxno);
  if  (f_taskidxno == NO_NEXTTASK)   goto  out_10;
      /* THIS IS THE LAST TASK ON THE SAME PROCESSOR AS THE CURRENT
         TASK */

  f_ltidno = get_list_tidno2(f_taskidxno);

  switch  (f_ltidno.execready)   {
    case   NOT_EXECREADY:
      mod_list_tidno2(f_taskidxno, EXECREADY);
      break;
    case   EXECREADY_WAIT:
      j = 0;
      while  ((j < ptr.wqexecready) &&
              (wqexecready[j].inmsg.tidxno != f_taskidxno))  j++;
      wqexecready[j].used_mark = 0;

      f_outmsg.u1.execready = EXECREADY;
      f_outmsg.u2.comm_mark = f_nothing;
      put_mq0(wqexecready[j].inmsg.tidno, f_outmsg);

      mod_list_tidno2(f_taskidxno, EXECREADY);
      break;
    default: /*** COULD NOT BE "EXECREADY" ***/
      break;
  }
```

```
out_10:
  i++; /*** null ***/
}



/*********************************************************************
  Operations on Message Queues
 *********************************************************************/

/*-------------------------------------------------------------------*
   Get message from the incoming message queue
 *-------------------------------------------------------------------*/
struct in_msg  get_mq1()
{
  struct  in_msg inmsg;


  pvm_recv(-1, MSGTYPE1);
  pvm_upkint(&inmsg.tidxno, 1, 1);
  pvm_upkint(&inmsg.tidno, 1, 1);
  pvm_upkint(&inmsg.event_type, 1, 1);
  pvm_upkint(&inmsg.u1.partidxno, 1, 1);
  pvm_upkint(&inmsg.u2.chdtidxno, 1, 1);

  return  inmsg;
}



/*-------------------------------------------------------------------*
   Send the message out on outgoing message queue
 *-------------------------------------------------------------------*/
put_mq0(int tidno, struct out_msg  outmsg)
{
  pvm_initsend(ENCODING);
  pvm_pkint(&outmsg.u1.tidno, 1, 1);
  pvm_pkint(&outmsg.u2.comm_mark, 1, 1);
  pvm_send(tidno, MSGTYPE0);
}
```

```
/*********************************************************************
  Operations on lists and waiting queues
 *********************************************************************/

/*-------------------------------------------------------------------*
   Get a record from list 'tidno', return 'tidno'
 *-------------------------------------------------------------------*/
int  get_list_tidno(int tidxno)
{
  int  i = 0, f_tid;


  while  (ltidno[i].tidxno != tidxno)   i++;
  switch  (ltidno[i].rec_mark)   {
    case  LOCKED:
      f_tid = LOCKED;
      break;
    case  NOT_PROCESSED:
      f_tid = NOT_PROCESSED;
      break;
    case  SPAWNED:
      f_tid = ltidno[i].tidno;
      break;
  }
  return  f_tid;
}



/*-------------------------------------------------------------------*
   Get a record from list 'tidno', return 'execready'
 *-------------------------------------------------------------------*/
struct list_tidno  get_list_tidno2(int  tidxno)
{
  int  i=0;

  while  (ltidno[i].tidxno != tidxno) i++;
  return  ltidno[i];
}



/*-------------------------------------------------------------------*
   Get a record from list 'commmark'
```

```
 *-------------------------------------------------------------*/
int  get_list_commmark(int partidxno, int chdtidxno)
{
  int  i = 0, f_commmark;


  while  (((lcommmark[i].partidxno != partidxno) ||
           (lcommmark[i].chdtidxno != chdtidxno))  &&
           (i < ptr.lcommmark))   i++;
  if  (i >= ptr.lcommmark)
    f_commmark = NOT_FOUND;
  else
    f_commmark = lcommmark[i].comm_mark;

  return  f_commmark;
}



/*-------------------------------------------------------------*
  - The calling functions could be: det_chdspawn, spawn
  - Modify the record of 'tidxno' --- only three fields actually:
    rec_mark, tidxno, tidno;  The last field 'curparnum' is only
    modified by the event 'nospawn'.
  - If there is a new task spawned (i.e. get the 'spawn' event),
    check the waiting queue 'wqtidno', to see whether there are
    other tasks waiting for this newly-spawned task.
 *-------------------------------------------------------------*/
mod_list_tidno(int rec_mark, int tidxno, int tidno)
{
  int  i, f_taskptr;
  struct  in_msg f_inmsg;
  struct  out_msg f_outmsg;


  i = 0;
  while  (ltidno[i].tidxno != tidxno)   i++;
  f_taskptr = i;

/*** MODIFY INTO LIST ltidno ***/
  ltidno[i].rec_mark = rec_mark;
  ltidno[i].tidxno = tidxno;
  ltidno[i].tidno = tidno;
```

```
/*** IF THIS IS THE NEWLY SPAWNED TASK, CHECK WAITING QUEUE
     wqtidno ***/
  if  (rec_mark == SPAWNED)   {        /*** ONLY EVENT: spawn ***/
    i = 0;
    while  (i<ptr.wqtidno)   {
      if  (wqtidno[i].used_mark == 1)   {
        f_inmsg = wqtidno[i].inmsg;
        if  (f_inmsg.u2.chdtidxno == tidxno)   {
          wqtidno[i].used_mark = 0;
          f_outmsg.u1.tidno = tidno;
          f_outmsg.u2.comm_mark = f_nothing;
          put_mq0(f_inmsg.tidno, f_outmsg);
        }
      }
      i++;
    }
  }
}



/*-----------------------------------------------------------------*
   Modify the field 'execready'
 *-----------------------------------------------------------------*/
mod_list_tidno2(int tidxno, int execready)
{
  int  i;

  i = 0;
  while  (ltidno[i].tidxno != tidxno)   i++;
  ltidno[i].execready = execready;
}



/*-----------------------------------------------------------------*
  - Affected events: send, nosend, nospawn
  - Put one record in list 'commmark' to express whether two tasks
    (parent and child) communicate or not.
  - Check waiting queue 'wqcommmark' to see whether there are any
    tasks which wait for messages from this parent task.
 *-----------------------------------------------------------------*/
put_list_commmark(int partidxno, int chdtidxno, int comm_mark)
```

```
{
  int  i;
  struct  in_msg f_inmsg;
  struct  out_msg f_outmsg;


/*** PUT INTO LIST lcommmark ***/
  for  (i=0; i<ptr.lcommmark; i++)
    if  ((lcommmark[i].partidxno == partidxno)  &&
   (lcommmark[i].chdtidxno == chdtidxno))   break;

  /*** no such record in list commmark ***/
  if  (i >= ptr.lcommmark)   {
    i = ptr.lcommmark++;
    lcommmark[i].partidxno = partidxno;
    lcommmark[i].chdtidxno = chdtidxno;
    lcommmark[i].comm_mark = comm_mark;
  }


/*** CHECK WAITING QUEUE wqcommmark ***/
  i = 0;
  while  (i<ptr.wqcommmark)   {
    if  (wqcommmark[i].used_mark == 1)   {
      f_inmsg = wqcommmark[i].inmsg;
      if  ((f_inmsg.u2.partidxno == partidxno) &&
           (f_inmsg.u1.chdtidxno == chdtidxno))   {
        wqcommmark[i].used_mark = 0;
        f_outmsg.u1.tidno = get_list_tidno(partidxno);
        f_outmsg.u2.comm_mark = comm_mark;
        put_mq0(f_inmsg.tidno, f_outmsg);
      }
    }
    i++;
  }
}



/*-------------------------------------------------------------*
  Put event into waiting queue 'tidno': such an event aims to detect
  the 'tidno' of a task, but 'tidno' has not been put into 'spawn'
  list yet.
 *-------------------------------------------------------------*/
```

```
put_wqueue_tidno(struct in_msg  inmsg)
{
  int  i = 0;

  while  ((wqtidno[i].used_mark == 1) && (i < ptr.wqtidno))  i++;
  wqtidno[i].used_mark = 1;
  wqtidno[i].inmsg = inmsg;
  if  (i >= ptr.wqtidno)   ptr.wqtidno = i + 1;
}



/*-----------------------------------------------------------------*
  Put event into waiting queue 'commmark' such event aims to get
  the information about whether a task has sent or has not sent data
  to another task, so that a 'recv' operation can be issued.
 *-----------------------------------------------------------------*/
put_wqueue_commmark(struct in_msg  inmsg)
{
  int  i = 0;

  while  ((wqcommmark[i].used_mark == 1) && (i < ptr.wqcommmark))
    i++;
  wqcommmark[i].used_mark = 1;
  wqcommmark[i].inmsg = inmsg;
  if  (i >= ptr.wqcommmark)  ptr.wqcommmark = i + 1;
}



/*-----------------------------------------------------------------*
  Put event into waiting queue 'execready': such event aims to get
  the information about whether a task is ready to go or not ---
  according to its execution sequence order determined by the
  scheduling policy.  Although some of the scheduling algorithms do
  not care about the execution sequence of tasks residing on the
  same processor, however, execution sequence does have significant
  influence on the task execution efficiency.
 *-----------------------------------------------------------------*/
put_wqueue_execready(struct in_msg  inmsg)
{
  int  i = 0;

  while  ((wqexecready[i].used_mark == 1) && (i < ptr.wqexecready))
```

```
    i++;
  wqexecready[i].used_mark = 1;
  wqexecready[i].inmsg = inmsg;
  if  (i >= ptr.wqexecready)  ptr.wqexecready = i + 1;
}



/*-----------------------------------------------------------------*
  Next task which is assigned on the same processor as the current
  task, and right after this current task.
 *-----------------------------------------------------------------*/
int  nexttask_torun(int  curtaskidxno)
{
  int  i, f_procidxno, f_execseqno, f_curtaskidxno, f_found;
  struct  list_tidno  f_ltidno;


  for  (i=0; policy[i].procidxno!=MAXNUM; i++)
    if  (policy[i].taskidxno == curtaskidxno)   break;

  if  (policy[i].procidxno == MAXNUM)   {
    error_handling("nexttask_torun");
    exit(1);
  }

  f_procidxno = policy[i].procidxno;
  f_execseqno = policy[i].execseqno;
  f_execseqno++;

  f_found = 0;
  while  (f_found == 0)   {
    for  (i=0; policy[i].procidxno!=MAXNUM; i++)
      if  ((policy[i].procidxno == f_procidxno) &&
           (policy[i].execseqno == f_execseqno))   break;

    if  (policy[i].procidxno == MAXNUM)   return  NO_NEXTTASK;

    f_curtaskidxno = policy[i].taskidxno;
    f_ltidno = get_list_tidno2(f_curtaskidxno);

    if  (f_ltidno.curparnum != 0)    return  f_ltidno.tidxno;
    else   f_execseqno++;
  }
```

```
}


/*-------------------------------------------------------------------*
  Put record into list 'ltexec', for keeping the record of task
  execution in the current run.
  - only the starting task having the 'partidxno' equal to
    'chdtidxno', and this record does not have to be put into the
    list.
 *-------------------------------------------------------------------*/
put_list_texec(int partidxno, int chdtidxno)
{
  if  (partidxno != chdtidxno)   {
    ltexec[ptr.ltexec].partidxno = partidxno;
    ltexec[ptr.ltexec].chdtidxno = chdtidxno;
    ptr.ltexec++;
  }
}
```

Appendix B

A Preprocessed *ATME* Parallel Task

This appendix provides an example of a user-provided parallel task, and its corresponding *ATME* preprocessed task. The user-provided task contains all three typical runtime operations: data reception, task spawn and data transmission. The code generated by *ATME* is delimited with comments, for the sake of clarity. The task model reflected in the program is given in Figure 3.2(a). Be aware that there exists an *execution monitor* which retains task runtime information and controls the task execution. Functions such as *det_parsend*, *det_chdspawn* and *put_spawn* are explained afterwards.

B.1 The Original Code

```
/*-----------------------------------------------------------------*
                A user-provided parallel task
  This task is named "A", which receives data from task "S", and
  spawns and sends data to tasks "B" and "D".
 *-----------------------------------------------------------------*/
#include       <stdio.h>
#include       <stdlib.h>
#include       "atme_0global.h"

main(int in_argc, char **in_argv)
{
  int          i, f_tidxno;
  char         f_str[100];

/* Initialize the out_argv array (storing spawning parameters);
 * The first three arguments are reserved for ATME use.  */
  for  (i=0; i<OUT_ARGV_NUM; i++)
    out_argv[i] = (char *)malloc(sizeof(char)*ARGV_LEN);

/* Receive data from task "S".  The "tidxno" of task S is in
```

```
 * in_argv[3], assigned when this task A is spawned by task S.
 * Recall the first three input parameters are reserved by ATME.    */
  tme_recv(in_argv[3], MSGTYPE2);
  tme_upkstr(f_str, 1, 1);

/* Do something */
  do_something(100);

/* Spawn and send data to task "B", if the condition is true.
 * cond1() simulates the conditions associated with the task spawning
 * operation between task A and B.  */
  if  (cond1() == 1)  {
    f_tidxno = tme_spawn("B", out_argv);
    if  (cond2() == 1)  {
      tme_initsend(0);
      strcpy(f_str, "A to B");       // The message is "A to B".
      tme_pkstr(f_str, 1, 1);
      tme_send(f_tidxno, MSGTYPE2);
    }
  }

/* Spawn and send data to task "D", if the conditions simulated by
 * cond3() is true. */
  if  (cond3() == 1)  {
    f_tidxno = tme_spawn("D", out_argv);
    if  (cond4() == 1)  {
      tme_initsend(0);
      strcpy(f_str, "A to D");       // The message is "A to D"
      tme_pkstr(f_str, 1 ,1);
      tme_send(f_tidxno, MSGTYPE2);
    }
  }


/* Exit ATME, but still exist as a process */
  tme_exit();
}
```

B.2 The ATME-Generated Code

```
/*-------------------------------------------------------------*
                    ATME-translated Code
  This task is named "A", which receives data from task "S", and
```

```
  spawns and sends data to tasks "B" and "D".
 *-------------------------------------------------------------------*/
#include       <stdio.h>
#include       <stdlib.h>
#include       "atme_0global.h"


/*** ATME code begins ***/
#include       "pvm3.h"
#include       "atme_0ext.h"
#include       "atme_0pa_global.h"
#include       "atme_0tmc_global.h"
/*** ATME code ends ***/


main(int in_argc, char **in_argv)
{
  int          i, f_tidxno;
  char         f_str[100];
  struct detect_info  detinfo;          // ATME structure.


/*** ATME code begins. ***/
  mytidno = pvm_mytid();

  estidno = stoi(in_argv[1]);           // in_argv[0]: task name.
  mytidxno = stoi(in_argv[2]);

  /* Load the task model, the processor model, and the scheduling
   * policy generated prior to program execution. */
  loadtasks();
  loadprocs();
  loadpolicy();

  /* Detect whether it is this task's turn to run, i.e., the control
   * of the execution commencement order of tasks distributd onto
   * the same processor. */
  detinfo = det_execready(mytidxno, mytidno);

  /* ATME-used variables */
  spawnno = 0;
  sendno = 0;
  recvno = 0;
```

```
  detinfo = det_execready(mytidxno, mytidno);
/*** ATME code ends. ***/


/* Initialize the out_argv array (storing spawning parameters);
 * The first three arguments are reserved for ATME use. */
  for  (i=0; i<OUT_ARGV_NUM; i++)
    out_argv[i] = (char *)malloc(sizeof(char)*ARGV_LEN);


/*** ATME-translated code begins. ***/
  /* Receive data from task "S".  The "tidxno" of task S is in
   * in_argv[3], given when this task A is spawned by task S.
   * Recall the first three input parameters are reserved by ATME. */
  partidxno = in_argv[3];

  /* Ask the "execution monitor" for the information about whether
   * there are any data transmitted from the parent task. */
  detinfo = det_parsend(mytidxno, mytidno, partidxno);

  /* If there is data sent off from its parent task S. */
  if  (detinfo.comm_mark == COMM)   {
    pvm_recv(detinfo.tidno, MSGTYPE2);
    pvm_upkstr(f_str, 1, 1);
    printf("A receives from S: %s\n", f_str);
  }
  else
    printf("A doesn't receive from S \n");
/*** ATME-translated code ends. ***/


/* Do something. */
  do_something(100);


/*** AMTE-translated code begins. ***/
  /* Spawn and send data to task "B", if the condition is true.
   * cond1() simulates the conditions associated with the operations
   * between task A and B.  */
  /* Detect and spawn task B. */
  // Get the task index number of the child task "B".
  chdtidxno = getchdtidxno(mytidxno, "B", spawnno++);
  if  (cond1() == 1)
    {
```

```
    detinfo = det_chdspawn(mytidxno, mytidno, chdtidxno);

    /* If the child task has not yet been spawned... */
    if  (detinfo.tidno == NOT_SPAWN)
      {
        /* Read the scheduling policy file to obtain the processor,
         * which is pre-defined by the ATME scheduling algorithm,
         * on which this task "B" can be allocated. */
        strcpy(procname, getproc(chdtidxno));

        /* Prepare parameters for spawning the child task.  The
         * first three parameters are reserved by ATME.  Note that
         * the out_argv[2] stores the task index number of the
         * current task, i.e., the parent task of its to-be-spawned
         * child task. */
        strcpy(out_argv[0], itos(estidno));
        strcpy(out_argv[1], itos(chdtidxno));
        strcpy(out_argv[2], itos(mytidxno));

        numt = pvm_spawn("B", out_argv, 1, procname,
                         NTASK1, &tidno1);
        if  (numt < NTASK1)
          error_handling("[A]: error in spawn B ...");

        tasks[chdtidxno].tidno = tidno1;

        /* Inform the "execution monitor" about current status of
         * task "A" and "B". */
        put_spawn(chdtidxno, tidno1);

        printf("A spawns B \n");
      }
    else
      tasks[chdtidxno].tidno = detinfo.tidno;

/* Send data to task B. */
    if  (cond2() == 1)
      {
        pvm_initsend(ENCODING);
        strcpy(f_str, "A to B");
        pvm_pkstr(f_str, 1, 1);
        pvm_send(tasks[chdtidxno].tidno, MSGTYPE2);

        put_send(mytidxno, chdtidxno);
```

```
          printf("A send to B: %s\n", f_str);
        }
      else
        {
          put_nosend(mytidxno, chdtidxno);
          printf("A doesn't send to B\n");
        }
    }
  else
    {
      put_nospawn(mytidxno, chdtidxno);
      printf("A doesn't spawn B\n");
    }


/* Spawn and send data to task "D", if the condition is true. */
  // Detect and spawn task D.
  /* Follow the same steps as those when spawning task B.
                  (listed above) */
      ...

  // Send data to task D.
  /* Follow the same steps as those when sending data to task B.
                  (listed above) */
      ...

  /* Inform the "execution monitor" that this task completes its
   * execution, and the next task allocated on the same processor
   * can commence to execute. */
  put_nextexec(mytidxno, in_argv[3], ++in_argv[4]);

  pvm_exit();
  exit(1);
/*** ATME-translated code ends. ***/
}
```

B.3 Functions Employed

```
Functionality of some functions employed above:
  error_handling: prompt fatal system error and stop the program.
  cond():         a function simulating condition(s) associated with
                  the task runtime operations.
```

```
do_something(): simulating the functionality behaviour of a task.

getchdtidxno:   get the "task index number" of the child task.
getpartidxno:   get the "task index number" of the parent task.
det_execready:  detect whether it is the turn of this task to run.
det_chdspawn:   send an "ev1_detchdspawn" event to the "execution
                monitor" task.
det_parsend:    send the event "ev1_detparsend" to the "execution
                monitor".

pvm_spawn:      PVM primitive handling task spawn.
pvm_initsend:   PVM primitive to initialize data transmission.
pvm_pkstr:      PVM primitive to pack data into the sending buffer.
pvm_upkstr:     PVM primitive to unpack received message.
pvm_send:       PVM primitive to undertake data transmission.
pvm_recv:       PVM primitive to receive a message from a task.

put_spawn:      send an "ev1_spawn" event to the "execution
                monitor" task.
put_nospawn:    send the an "ev1_nospawn" event to the "execution
                monitor" task.
put_send:       send the "ev1_send" event to "execution monitor".
put_nosend:     send the "ev1_nosend" event to "execution monitor".
```

Bibliography

[1] Thomas L. Adam, K. M. Chandy and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, Volume 17, Number 12, pages 685–690, December 1974.

[2] Shakil Ahmed, Nicholas Carriero and David Gelernter. A programming building tool for parallel applications. Technical Report UT-CS-205, Department of Computer Science, Yale University, 1993.

[3] Kento Aida, Hironori Kasahara and Seinosuke Narita. Job scheduling scheme for pure space sharing among rigid jobs. In D. G. Feitelson and L. Rudolph (editors), *Job Scheduling Strategies for Parallel Processing, Lecture Notes of Computer Science, Volume 1459*, pages 98–121. Springer-Verlag, 1998.

[4] R. Allan, D. Baumgartner, K. Kennedy and A. Porterfield. PTOOL: A semiautomatic parallel programming assistant. In *Proceedings of International Conference on Parallel Processing*, pages 164–170, August 1986.

[5] G. M. Amdahl. Validity of the single-processor approach of achieving large scale computing capabilities. In *AFIPS Conference Proceedings, Volume 30*, pages 483–485, Atlantic City, N.J., April 1967.

[6] George A. Anderson and E. Douglas Jensen. Computer interconnection structures: Taxonomy, characteristics, and examples. *ACM Computing Surveys*, Volume 7, Number 4, pages 197–213, December 1975.

[7] G. R. Andrews, D. P. Dobkin and P. J. Downey. Distributed allocation with pools of servers. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 73–83, August 1982.

[8] F. D. Anger, J. J. Hwang and Y. C. Chow. Scheduling with sufficient loosely coupled processors. *Journal of Parallel and Distributed Computing*, Volume 9, Number 1, pages 87–92, 1990.

[9] W. F. Appelbe and C. McDowell. Anomaly detection in parallel fortran programs. In *Proceedings of Workshop on Parallel Processing Using HEP*, May 1985.

[10] R. A. Baeza-Yates, G. Quezada and G. Valmadre. Visual debugging and automatic animation of C programs. In Peter Eades and Kang Zhang (editors), *Software Visualisation, Series on Software Engineering and Knowledge Engineering, Volume 7*, pages 46–58. World Scientific, 1996.

[11] J. A. Bannister and K. S. Trivedi. Task allocation in fault-tolerant distributed system. *Acta Informatica*, Volume 20, Number 3, pages 261–281, December 1983.

[12] Mokhtar S. Bazaraa, John J. Jarvis and Hanif D. Sherali. *Linear programming and network flows.* John Wiley and Sons Inc., second edition, 1990.

[13] William D. Becher and Eric M. Aupperle. The communications hardware of the merit computer network. *IEEE Transactions on Communications*, Volume COM-20, Number 3, pages 516–526, June 1972.

[14] Adam Beguelin. Xab: A tool for monitoring pvm programs. In *Workshop on Heterogeneous Processing*, pages 92–97, Los Alamitos, California, April 1993. IEEE Computer Society Press.

[15] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Keith Moore, Peter Newton and Vaidy Sunderam. HeNCE: A user's guide (version 2.0). Technical Report UT-CS-205, Department of Computer Science, The University of Tennessee, 1993.

[16] F. Berman and L. Synder. On mapping parallel algorithms into parallel architectures. In *International Conference on Parallel Processing*, pages 307–309, August 1984.

[17] A. Billionnet, M. C. Costa and A. Sutte. An efficient algorithm for a task allocation problem. *Journal of the Association for Computing Machinery*, Volume 39, Number 3, pages 502–518, 1992.

[18] S. Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Transactions of Software Engineer*, Volume SE-7, Number 6, pages 583–589, 1981.

[19] S. H. Bokhari. Dual processor scheduling with dynamic reassignment. *IEEE Transactions on Software Engineering*, Volume SE-5, Number 4, pages 326–334, July 1979.

[20] Allan Borodin, Nathan Linial and Michael E. Saks. An optimal on-line algorithm for metrical task system. *Journal of the Association for Computing Machinery*, Volume 39, Number 4, pages 745–763, October 1992.

[21] David C. C. Bover, Kevin J. Maciunas and Michael J. Oudshoorn. *Ada: A First Course in Programming and Software Engineering.* International Computer Science Series. Addison-Wesley, Sydney, 1992.

[22] James C. Browne, Syed I. Hyder, Jack Dongarra, Keith Moore and Peter Newton. Visual programming and debugging for parallel computing. *IEEE Parallel and Distributed Technology*, Volume 3, Number 1, pages 335–355, Spring 1995.

[23] J. L. Bruno. Deterministic and stochastic scheduling problem with treelike precedence constraints. In M. A. H. Dempster, J. K. Lenstra and A. H. G. Rinnooy Kan (editors), *Deterministic and Stochastic Scheduling*, pages 367–374. Reidel, Dordrecht, 1982.

[24] Peter Buhler. The COIN model for concurrent computation and its implementation. *Microprocessing and Microprogramming*, Volume 30, Number 1-5, pages 577–584, August 1990.

[25] Peter Buhler and Dieter Wybranietz. Tools for distributed programming in the INCAS project. *Microprocessing and Microprogramming*, Volume 27, Number 1-5, pages 199–206, 1989.

[26] Alan Burns and Andy Wellings. *Concurrency in ADA*. Cambridge University Press, Cambridge, New York, 1995.

[27] Ralph M. Butler and Ewing L. Lusk. User's guide to the P4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, Argonne, IL., 1992.

[28] Ralph M. Butler and Ewing L. Lusk. Monitors, messages, and clusters: the P4 parallel programming system. *Parallel Computing*, Volume 20, Number 4, pages 547–564, April 1994.

[29] Luis-Felipe Cabrera, Edward Hunter, Michael J. Karels and David A. Mosher. User-process communication performance in networks of computers. *IEEE Transactions on Software Engineering*, Volume 14, Number 1, pages 38–53, January 1988.

[30] W. T. Cai, W. J. Milne and S. J. Turner. Graphical views of the behaviour of parallel programs. *Journal of Parallel and Distributed Computing*, Volume 18, Number 2, pages 223–230, 1993.

[31] W. T. Cai, H. K. Tan and S. J. Turner. Visual programming for parallel processing. In Peter Eades and Kang Zhang (editors), *Software Visualisation, Series on Software Engineering and Knowledge Engineering, Volume 7*, pages 119–140. World Scientific, 1996.

[32] W. T. Cai and S. J. Turner. Process scheduling and program monitoring on transputers. In S. Atkins and A. S. Wagner (editors), *Transputer Research and Applications, NATUG-6, Proceedings of the 6th Conference of the North American Transputer Users Group*, pages 290–305. ISO Press, 1993.

[33] W. T. Cai and S. J. Turner. An approach to the run-time monitoring of parallel programs. *The Computer Journal*, Volume 37, Number 4, pages 333–345, 1994.

[34] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, Volume 32, Number 4, pages 444–458, April 1989.

[35] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, Volume 14, Number 2, pages 141–154, February 1988.

[36] L. M. Casey. Decentralized scheduling. *Australian Computer Journal*, Volume 13, pages 58–63, May 1981.

[37] N. F. Chen and C. L. Liu. On a class of scheduling algorithms for multiprocessor computing systems. In T. Y. Feng (editor), *Lecture Notes in Computer Science*, pages 1–16. Springer, New York, 1975.

[38] Jong Deok Choi, Ron Cytron and Jeanne Ferrante. On the efficient enginerring of ambitious program analysis. *IEEE Transactions on Software Engineering*, Volume 20, Number 2, pages 105–114, February 1994.

[39] T. C. K. Chou and J. A. Abraham. Load balancing in distributed systems. *IEEE Transactions on Software Engineering*, Volume SE-8, Number 4, pages 401–412, July 1982.

[40] Y. C. Chow and W. H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transactions on Computers*, Volume C-28, Number 5, pages 354–361, May 1979.

[41] W. W. Chu, D. Lee and B. Iffla. A distributed processing system for naval data communication networks. In *AFIPS Conference Proceedings, Volume 47, NCC*, pages 783–793, 1978.

[42] Wesley W. Chu, Leslie J. Holloway, Min-Tsung Lan and Kemal Efe. Task allocation in distributed data processing. *Computer*, Volume 13, Number 11, pages 57–69, November 1980.

[43] E. G. Coffman Jr. *Computer and Job-Shop Scheduling Theory.* John Wiley, New York, 1976.

[44] E. G. Coffman Jr. and R. L. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, Volume 1, Number 3, pages 200–213, 1972.

[45] The PORTS Consortium. The PORTS0 interface. version 0.3. Technical Report ANL/MCS-TM-203, Argonne National Laboratory, February 1995.

[46] Greg Costello. *Some applications of regression analysis for the purpose of real estate market research.* Curtin University of Technology, Perth, West Australia, 1993.

[47] W. Crowther. Performance measurements on a 128-node butterfly parallel processor. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 531–535, 1985.

[48] Ron Cytron, Michael Hind and Wilson Hsieh. Automatic generation of dag parallelism. In *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, pages 54–68, Portland, Oregan, USA, June 21-23 1989. SIGPLAN, Volume 24, Number 7, July 1989.

[49] C. R. Dow, S. K. Chang and M. L. Soffa. Visual transformation specifications. In Peter Eades and Kang Zhang (editors), *Software Visualisation, Series on Software Engineering and Knowledge Engineering, Volume 7*, pages 141–162. World Scientific, 1996.

[50] T. H. Dunigan. Performance of the intel ipsc/860 and ncube 6400 hypercube. Technical Report ORNL/TM-11790, Oak Ridge National Laboratory, TN, 1991.

[51] Hesham El-Rewini and Hesham H. Ali. Static scheduling of conditional branches in parallel programs. *Journal of Parallel and Distributed Computing*, Volume 24, Number 1, pages 41–54, January 1995.

[52] Hesham El-Rewini, Hesham H. Ali and Ted Lewis. Task scheduling in multiprocessing systems. *Computer*, Volume 28, Number 12, pages 27–37, December 1995.

[53] Hesham El-Rewini and Ted G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, Volume 9, Number 2, pages 138–153, June 1990.

[54] D. J. Kuck et.al. Measurement of parallelism in ordinary Fortran programs. In *Proceedings of Sagamore Conference on Parallel Processing*, pages 23–36, 1973.

[55] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, Volume 16, Number 4, pages 308–316, 1992.

[56] Norman Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, Volume 20, Number 3, pages 199–206, March 1994.

[57] Robert O. Ferguson and Lauren F. Sargent. *Linear programming: Fundamentals and applications.* McGraw-Hill Book Company, Inc., 1958.

[58] E. B. Fernandez and B. Bussell. Bounds on the number of processors and time for multiprocessor optimal schedules. *IEEE Transactions on Computers*, Volume C-22, Number 8, pages 745–751, August 1973.

[59] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Lamguages and Systems*, Volume 9, Number 3, pages 319–349, July 1987.

[60] Horace P. Flatt and Ken Kennedy. Performance of parallel processors. *Parallel Computing*, Volume 12, Number 1, pages 1–20, October 1989.

[61] Michael J. Flynn. Very high speed computing systems. *Proceedings of the IEEE*, Volume 54, Number 12, pages 1901–1909, December 1966. Special issues on Computer.

[62] J. A. B. Fortes and F. Parisi-Presicce. Optimal linear schedules for the parallel execution of algorithms. In *1984 International Conference on Parallel Processing*, pages 322–329, August 1984.

[63] Ian Foster. Compositional C++. In *Design and Building Parallel Programs (Online)*, Chapter 5. Addison-Wesley Inc., Argonne National Laboratory and the NSF Center for Research on Parallel Computation. URL: http://www.mcs.anl.gov/dbpp/ax.

[64] Ian Foster. Automatic generation of self-scheduling programs. *IEEE Transactions on Parallel and Distributed Systems*, Volume 2, Number 1, pages 68–78, January 1991.

[65] B. Furht. A contribution to classification and evaluation of structures for parallel computers. *Microprocessing and Microprogramming*, Volume 25, pages 203–208, 1989.

[66] A. Gabrielian and D. B. Tyler. Optimal object allocation in distributed computer systems. In *Proceedings of 4th International Conference on Distributed Computing Systems*, pages 84–95, May 1984.

[67] D. Gannon and J. Van Rosendale. On the impact of communication complexity in the design of parallel algorithms. Technical Report 84-41, ICASE, 1984.

[68] J. L. Gaudiot, J. I. PI and M. L. Campbell. Program graph allocation in distributed multicomputers. *Parallel Computing*, Volume 7, pages 227–247, 1988.

[69] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek and Vaidy Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Cambridge, Massachusetts, 1994.

[70] T. Gonzalez and S. Sahni. Flowshop and jobshop schedules: Complexity and approximation. *Operations Research*, Volume 26, Number 1, pages 36–52, January–February 1978.

[71] Brent Gorda and Rich Wolski. Time sharing massively parallel machines. In *Proceedings of 1995 International Conference on Parallel Processing, Volume 2*, pages 214–217, 1995.

[72] R. L. Graham. Bounds on multiprocessing time anomalies. *SIAM Journal on Applied Mathematics*, Volume 17, Number 2, pages 416–429, March 1969.

[73] R. L. Graham, E. K. Lawler, J. K. Lenstra and H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In P. L. Hammer, E. L. Johnson and B. H. Korte (editors), *Discrete Optimization II, Volume 5*, pages 287–326. 1979.

[74] Peter J. Green. *Nonparametric regression and generalized linear models: a roughness penalty approach.* Chapman and Hall, London, 1994.

[75] Anne Greenbaum. Synchronization costs on multiprocessors. *Parallel Computing*, Volume 10, Number 1, pages 3–14, March 1989.

[76] J. L. Gustafson. Reevaluating amdahl's law. *Communication of the ACM*, Volume 31, Number 5, pages 532–533, May 1988.

[77] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174. ACM New York, January 1992.

[78] F. Harary. *Graph theory.* Addison-Wesley, New York, 1969.

[79] Alfred C. Hartmann. *A Concurrent PASCAL Compiler for Minicomputers*, Volume 50 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, Berlin, 1977.

[80] Leonard S. Haynes, Richard L. Lau, Daniel P. Siewiorek and David W. Mizell. A survey of highly parallel computing. *Computer*, Volume 15, Number 1, pages 9–24, January 1982.

[81] Ulrich Herzog. Performance evaluation principles for vector- and multiprocessor systems. *Parallel Computing*, Volume 7, Number 3, pages 425–438, September 1988.

[82] Ronald R. Hocking. *Methods and applications of linear models : regression and the analysis of variance.* John Wiley, New York, 1996.

[83] R. W. Hockney. $(r_\infty, n_{1/2}, s_{1/2})$ measurements on the 2-CPU CRAY X-MP. *Parallel Computing*, Volume 2, Number 1, pages 1–14, March 1985.

[84] Roger W. Hockney and Ian J. Curington. $f_{1/2}$: A parameter to characterize memory and communication bottleneck. *Parallel Computing*, Volume 10, Number 3, pages 277–286, May 1989.

[85] R. Hofmann, R. Klar, B. Mohr an A. Quick and M. Siegle. Distributed performance monitoring: Methods, tools and applications. *IEEE Transactions on Parallel and Distributed Systems*, Volume 5, Number 6, pages 585–598, June 1994.

[86] Atsushi Hori, Hiroshi Tezuka and Yutaka Ishikawa. Overhead analysis of preemptive gang scheduling. In D. G. Feitelson and L. Rudolph (editors), *Job Scheduling Strategies for Parallel Processing, Lecture Notes of Computer Science, Volume 1459*, pages 217–230. Springer-Verlag, 1998.

[87] C. D. Howe and B. Moxon. How to program parallel computers. *IEEE Spectrum*, Volume 20, Number 9, pages 36–41, September 1987.

[88] Wilson C. Hsieh. Extracting parallelism from sequential programs. Technical report, Massachusetts Institute of Technology, May 1988.

[89] Y. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, Volume 9, pages 841–848, 1961.

[90] Lin Huang and Michael J. Oudshoorn. An approach to distribution of parallel programs with conditional task attributes. Technical Report TR97-06, Department of Computer Science, University of Adelaide, August 1997.

[91] Lin Huang and Michael J. Oudshoorn. ATME: A parallel programming environment for applications with conditional task attributes. In Andrzej Goscinski, Michael Hobbs and Wanlei Zhou (editors), *1997 3rd International Conference on Algorithms and Architectures for Parallel Processing*, pages 275–282, December 1997. Melbourne, Australia.

[92] Jing Jang Hwang. *Deterministic Scheduling in Systems with Interprocessor Communication Times.* Ph.D. thesis, Computer and Information Sciences Department, University of Florida, 1987.

[93] Jing Jang Hwang, Yuan Chieh Chow, Frank D. Anger and Chung Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, Volume 18, Number 2, pages 244–257, April 1989.

[94] Kamal Kumar Jain and V. Rajaraman. Lower and upper bounds on time for multiprocessor optimal schedules. *IEEE Transactions on Parallel and Distributed Systems*, Volume 5, Number 8, pages 879–886, August 1994.

[95] C. J. Jenny. Process partitioning in distirbuted systems. *Digest of Papers NTC'77*, Number 31, pages 1:1–1:10, 1977.

[96] Morris A. Jette. Expanding symmetric multiprocessor capability through gang scheduling. In D. G. Feitelson and L. Rudolph (editors), *Job Scheduling Strategies for Parallel Processing, Lecture Notes of Computer Science, Volume 1459*, pages 199–216. Springer-Verlag, 1998.

[97] Erice E. Johnson. Completing an mimd multiprocessor taxonomy. *Computer Architecture News*, Volume 16, Number 3, pages 44–47, June 1988.

[98] Erice E. Johnson. A prototype virtual port memory multiprocessor. Technical Report NMSU-ECE-88-003, New Mexico State University, Las Cruces, 1988.

[99] D. Karger and C. Stein. An $\tilde{o}(n^2)$ algorithm for minimum cuts. In *Proceedings of the 25th ACM Symposium on the Theory of Computing*, pages 757–765. ACM New York, May 1993.

[100] A. Karp. Programming for parallelism. *Computer*, Volume 20, Number 5, pages 43–57, May 1987.

[101] Dongseung Kim and Joonyoung Park. Two-way dominant sequence clustering for processor scheduling. *Information Processing Letters*, Volume 49, Number 4, pages 203–208, February 1994.

[102] S. J. Kim and J. C. Browne. A general approach to mapping parallel computation upon multiprocessor architectures. In *Proceedings of International Conference on Parallel Processing, Volume 3*, pages 1–8, 1988.

[103] David G. Kleinbaum and Lawrence L. Kupper. *Applied regression analysis and other multivariable methods.* Duxbury Press, North Scituate, Massachusetts, 1978.

[104] A. Kolawa. The Express programming environment. In *Workshop on Heterogeneous Network-Based Concurrent Computing*, Tallahassee, October 1991.

[105] Venkat Konda and Anup Kumar. A systematic framework for the dependence cycle removal in practical loops. *Journal of Parallel and Distributed Computing*, Volume 27, Number 2, pages 157–171, June 1995.

[106] E. Kraemer and J. T. Stasko. The visualization of parallel systems: An overview. *Journal of Parallel and Distributed Computing*, Volume 18, Number 2, pages 105–117, 1993.

[107] Phillip Krueger and Niranjan G. Shivaratri. Adaptive location policies for global scheduling. *IEEE Transactions on Software Engineering*, Volume 20, Number 6, pages 432–444, June 1994.

[108] Sukhamay Kundu. The call-return tree and its application to program performance analysis. *IEEE Transactions on Software Engineering*, Volume SE-12, Number 11, pages 1096–1098, November 1986.

[109] Swamy Kutti. Taxonomy of parallel processing and definitions. *Parallel Computing*, Volume 2, Number 4, pages 353–359, December 1985.

[110] Oak Ridge National Laboratory. Listing of new features in PVM 3.4.0 and changes in past versions. URL: http://www.epm.ornl.gov/pvm/changes.html, Computer Science and Mathematics Division.

[111] Oak Ridge National Laboratory. PVM supported architectures/OSs. URL: http://www.epm.ornl.gov/pvm/pvmArch.html, Computer Science and Mathematics Division.

[112] B. J. Lageweg, J. K. Lanstra and A. H. G. Rinnooy Kan. Jobshop scheduling by implicit enumeration. *Management Science*, Volume 24, Number 4, pages 441–450, 1977.

[113] S. Lam and R. Sethi. Worst case analysis of two scheduling algorithms. *SIAM Journal of Computing*, Number 6, pages 518–536, 1977.

[114] E. L. Lawler, J. K. Lenstra and A. H. G. Rinnooy Kan. Recent development in deterministic sequencing and scheduling: a survey. In M H. Dempster, J. K. Lenstra and A. H. G. Rinnooy Kan (editors), *Deterministic and Stochastic Scheduling*, pages 367–374. D. Reidel, Dordrecht, the Netherlands, 1982.

[115] Chung Yee Lee, Jing Jang Hwang, Yuan Chieh Chow and Frank D. Anger. Multiprocessor scheduling with interprocessor communication delays. *Operations Research Letters*, Volume 7, Number 3, pages 141–147, June 1988.

[116] Ted Lewis and Hesham El-Rewini. Parallax: A tool for parallel program scheduling. *IEEE Parallel and Distributed Technology*, Volume 1, Number 2, pages 63–73, May 1993.

[117] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. In *Proceedings of 4th International Conference on Distributed Computing Systems*, pages 30–39, May 1984.

[118] V. M. Lo. Task assignment to minimize completion time. In *5th International Conference on Distributed Computing Systems*, pages 329–336, May 1985.

[119] J. E. Lumpp, T. L. Casavant, J. A. Gannon, K.J. Williams and M. S. Andersland. Trace recovery for debugging parallel and distributed systems. In *The 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 208–210, San Diego, May 1993.

[120] P. Y. R. Ma, E. Y. S. Lee and J. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, Volume C-31, Number 1, pages 41–47, January 1982.

[121] Message Passing Interface Forum. MPI: A message-passing interface standard. URL: http://www.mcs.anl.gov/mpi/mpi-report, May 5 1994.

[122] M. Minsky and S. Papert. Associative information techniques. In E. J. Jacks (editor), *On Some Associative, Parallel and Analog Computations.* Elsevier, New York, 1971.

[123] Thomas J. Mowbray and Ron Zahavi. *The Essential CORBA: System Integration Using Distributed Objects.* John Wiley & Sons and the Object Management Group, July 1995.

[124] Dieter Müller-Wichards. Performance estimates for applications: An algebraic framework. *Parallel Computing*, Volume 9, pages 77–106, 1989.

[125] Randolph Nelson, Don Towsley and Asser N. Tantawi. Performance analysis of parallel processing systems. *IEEE Transactions on Software Engineering*, Volume 14, Number 4, pages 532–540, April 1988.

[126] L. M. Ni and K. Abani. Nonpreemptive load balancing in a class of local area networks. In *Proceedings of Computing Networking Symposium*, pages 113–118, December 1981.

[127] L. M. Ni and K. Hwang. Optimal load balancing for a multiple processor system. In *Proceedings on International Conference on Parallel Processing*, pages 352–357, 1981.

[128] Michael G. Norman and Peter Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, Volume 25, Number 3, pages 263–302, September 1993.

[129] Object Management Group. The common object request broker: Architecture and specification (CORBA). Object Management Group (OMG), Framingham, MA., Revision 1.2, Draft 29, December 1993.

[130] Object Management Group. The common object request broker: architecture and apecification (revision 2.0). Object Management Group (OMG), Framingham, Massachusetts, July 1995.

[131] Matthew T. O'Keefe and Henry G. Dietz. Static barrier MIMD: Architecture and performance analysis. *Journal of Parallel and Distributed Computing*, Volume 25, Number 2, pages 126–132, March 1995.

[132] Robert Olson. Parallel processing in a message-based operating system. *IEEE Software*, Volume 2, Number 4, pages 39–49, July 1985.

[133] Randy Otte, Paul Patrick and Mark Roy. *Understanding CORBA.* Prentice-Hall, 1995.

[134] K. J. Ottenstein. *Data-Flow Graphs as an Intermediate Program Form.* Ph.D. thesis, Computer Science Department, Purdue University, Lafayette, Indiana, August 1978.

[135] K. J. Ottenstein. An intermediate program form based on a cyclic data-dependence graph. Technical Report CS-TR 81-1, Department of Computer Science, Michigan Technological University, Houghton, Michigan, October 1981. July 1982 errata.

[136] Michael J. Oudshoorn and Lin Huang. Conditional task scheduling on loosely-coupled distributed processors. In *The 10th International Conference on Parallel and Distributed Computer Systems*, pages 136–140, October 1997. New Orleans, USA.

[137] Michael J. Oudshoorn, Hendra Widjaja and Sharon K. Ellershaw. Aspects and taxonomy of program visualisation. In Peter Eades and Kang Zhang (editors), *Software Visualisation, Series on Software Engineering and Knowledge Engineering, Volume 7*, pages 9–26. World Scientific, 1996.

[138] J. Ousterhout, D. Scelza and P. Sindhu. Medusa: An experiment in distributed operating system structure. *Communication on ACM*, Volume 23, Number 2, pages 92–105, Feburary 1980.

[139] D. A. Padua, D. J. Kuck and H. Lawrie D. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, Volume C-29, Number 9, pages 763–776, September 1980.

[140] Michael J. Panik. *Linear programming: mathematics, theory and algorithms.* Dordrecht; Boston: Kluwer Academic, 1996.

[141] J. K. Peir and D. D. Gajski. CAMP: A programming aid for multiprocessors. In *Proceedings of International Conference on Parallel Processing*, pages 475–482, August 1986.

[142] Dar Tzen Peng and Kang G. Shin. Optimal scheduling of cooperative tasks in a distributed system using an enumerative method. *IEEE Transactions on Software Engineering*, Volume 19, Number 3, pages 253–267, March 1993.

[143] C. C. Price and S. Krishnaprasad. Software allocation models for distributed computing systems. In *Proceedings of 4th International Conference on Distributed Computing Systems*, pages 40–48, May 1984.

[144] K. V. S. Ramarao and S. Venkatesan. The lower bounds on distributed shortest paths. *Information Processing Letters*, Volume 48, Number 3, pages 145–149, November 1993.

[145] V. J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. Internal Report C80-06, School of Information Systems, University of East Anglia, Norwich, 1986.

[146] Celso Ribeiro. Performance evaluation of vector implementations of combinatorial algorithms. *Parallel Computing*, Volume 1, Number 3-4, pages 287–294, December 1984.

[147] J. B. G. Roberts, J. G. Harp, B. C. Merrifield, K. J. Palmer, P. Simpson, J. S. Ward and H. C. Webber. Evaluating parallel processors for real-time applications. *Parallel Computing*, Volume 8, pages 245–254, 1988.

[148] Enders Anthony Robinson. *Least squares regression analysis in terms of linear algebra.* Houston, Texas, Goose Pond Press, 1981.

[149] James Snowdon Roper. *PL1 in Easy Stages: A Programmed Learning Textbook.* Elek Science, London, 1973.

[150] Vivek Sarkar. Determining average program execution times and their variance. *Proceedings of 1989 SIGPLAN Notices, Conference on Programming Language Design and Implementation*, Volume 24, Number 7, pages 298–312, July 1989.

[151] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* The MIT Press, Cambridge, MA, 1989.

[152] Uwe Schwiegelshohn and Ramin Yahyapour. Improving first-come-first-serve job scheduling by gang scheduling. In D. G. Feitelson and L. Rudolph (editors), *Job Scheduling Strategies for Parallel Processing, Lecture Notes of Computer Science, Volume 1459*, pages 180–198. Springer-Verlag, 1998.

[153] Chien Chung Shen and Wen Hsiang Tasi. A graph matching approach to optimal task assignment in distributed computing systems using a minmax criterion. *IEEE Transactions on Computers*, Volume C-34, Number 3, pages 197–203, March 1985.

[154] Warren Smith, Ian Foster and Valerie Taylor. Predicting application run times using historical information. In D. G. Feitelson and L. Rudolph (editors), *Job Scheduling Strategies for Parallel Processing, Lecture Notes of Computer Science, Volume 1459*, pages 122–142. Springer-Verlag, 1998.

[155] L. Snyder. Parallel programming and the POKER programming environment. *IEEE Computing*, pages 27–36, July 1984.

[156] V. A. Sposito. *Linear and nonlinear programming.* The Iowa State University Press, AMES, 1975.

[157] James H. Stapleton. *Linear statistical models.* Wiley, New York, 1995.

[158] Robert Stansbury Stockton. *Introduction to linear programming.* Allyn and Bacon Inc., Boston, second edition, 1963.

[159] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the Association for Computing Machinery*, Volume 44, Number 4, pages 585–591, July 1997.

[160] H. S. Stone and S. H. Bokhari. Control of distributed processes. *Computer*, Volume 11, pages 97–106, 1978.

[161] Harold S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, Volume SE-3, Number 1, pages 85–93, January 1977.

[162] Tony T. Y. Suen and Johnny S. K. Wong. Efficient task migration algorithm for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, Volume 3, Number 4, pages 488–499, July 1992.

[163] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, Volume 2, Number 4, pages 315–339, December 1990.

[164] V. S. Sunderam, G. A. Geist, J. Dongarra and R. Manchek. The PVM concurrent computing system: Evolution, experiences and trends. *Parallel Computing*, Volume 20, Number 4, pages 531–545, April 1994.

[165] Kuniyasu Suzaki and David Walsh. Implementing the combination of time sharing and space sharing on AP/Linux. In D. G. Feitelson and L. Rudolph (editors), *Job Scheduling Strategies for Parallel Processing, Lecture Notes of Computer Science, Volume 1459*, pages 83–97. Springer-Verlag, 1998.

[166] Clive Temperton. Further measurements of $(r_{\infty},n_{1/2})$ on the CRAY-1 and CRAY X-MP. *Parallel Computing*, Volume 11, Number 1, pages 107–111, July 1989.

[167] Ten H. Tzen and Lionel M. Ni. Dependence uniformization: A loop parallelization technique. *IEEE Transactions on Parallel and Distributed Systems*, Volume 4, Number 5, pages 547–558, May 1993.

[168] J. Ullman. NP-Complete scheduling problems. *Journal of Computing System Science*, Volume 10, Number 3, pages 384–393, June 1975.

[169] Steven Vajda. *Linear Programming: Algorithms and Applications.* Chapman and Hall, London, 1981.

[170] R. A. Wagner and K. S. Trivedi. Hardware configuration selection through discretizing a continuous variable solution. In *Proceedings of 7th IFIP Symposium on Computing Performance Modelling, Measurement and Evaluation*, pages 127–142, 1980. Toronto, Canada.

[171] D. Walsh, B. B. Zhou, C. W. Johnson and K. Suzaki. The implementation of a scalable gang scheduling scheme on the AP1000+. In *Proceedings of the 8th International Parallel Computing Workshop, Singapore*, pages P1:G1 – P1:G6, September 1998.

[172] R. C. Waters. Automatic analysis of the logical structure of programs. Technical Report TR-492, AI-Lab, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1978.

[173] Sanford Weisberg. *Applied linear regression.* Wiley, New York, 1980.

[174] John Werth, James C. Browne, Steve Sobek, T. J. Lee, Peter Newton and Ravi Jain. The interaction of the formal and the practical in parallel programming environment development: CODE. Technical Report TR-91-09, Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712-1188, April 1991.

[175] S. White, A. ålund and V. S. Sunderam. Performance of the NAS parallel benchmarks on PVM-based networks. *Journal of Parallel and Distributed Computing*, Volume 26, Number 1, pages 61–71, April 1995.

[176] Hendra Widjaja. Visor++: A software visualisation tool for task-parallel object-oriented programs. Master's thesis, Department of Computer Science, University of Adelaide, March 1998.

[177] Steven C. Wohlever. *Concurrency Control in a Dynamic Real-Time Distributed Object Computing Environment.* Ph.D. thesis, Computer Science Department, University of Rhode Island, 1997.

[178] Yaron Wolfstahl. Mapping parallel programs to multiprocessors: A dynamic approach. *Parallel Computing*, Volume 10, Number 1, pages 45–50, March 1989.

[179] Min You Wu and Daniel Gajski. Hypertool: a programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, Volume 1, Number 3, pages 330–343, July 1990.

[180] Tao Yang. *Scheduling and Code Generation for Parallel Architectures.* Ph.D. thesis, Department of Computer Science, Rutgers, The State University of New Jersey, 1993.

[181] Zhonghua Yang and Keith Duddy. CORBA: A platform for distributed object computing. *ACM Operating Systems Review*, Volume 30, Number 2, pages 4–31, April 1996.

[182] D. Q. Zhang, K. Zhang and J. Cao. Visual programming for heterogeneous distributed systems. In Peter Eades and Kang Zhang (editors), *Software Visualisation, Series on Software Engineering and Knowledge Engineering, Volume 7*, pages 163–182. World Scientific, 1996.

[183] Da Qian Zhang and Kang Zhang. A visual programming environment for distributed systems. In *Proceedings VL'95: 11th IEEE International Symposium on Visual Language*, pages 310–317. IEEE Computer Society Press, September 1995. Darmstadt, Germany.

[184] B. B. Zhou, R. P. Brent, D. Walsh and K. Suzaki. Job scheduling strategies for networks of workstations. In D. G. Feitelson and L. Rudolph (editors), *Job Scheduling Strategies for Parallel Processing, Lecture Notes of Computer Science, Volume 1459*, pages 143–157. Springer-Verlag, 1998.

[185] Weiping Zhu, Tyng-Yeu Liang and Ce-Kuen Shieh. Optimal task clustering using Hopfield net. In Andrzej Goscinski, Michael Hobbs and Wanlei Zhou (editors), *Proceedings of 1997 3rd International Conference on Algorithms and Architectures for Parallel Processing*, pages 451–464, December 1997.