A Parallel List Scheduling Algorithm: Design and Performance

by

Amin Arshad AbdulGhani

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

April, 1993

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 1354055

A parallel list scheduling algorithm: Design and performance

Abdulghani, Amin Arshad, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A PARALLEL LIST SCHEDULING ALGORITHM : DESIGN AND PERFORMANCE

BY

AMIN ARSHAD ABDULGHANI

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

APRIL 1993

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by **AMIN ARSHAD ABDULGHANI** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE in COMPUTER SCIENCE** .

THESIS COMMITTEE

Dr. M. Bozyigit (Chairman)

Dr. S. Ghanta (Member)

Dr. Meng Er (Member).

Department Chairman

Dean, College of Graduate Studies

14 - 4 - 1993

Date

Dedicated to

My Parents

and

Brothers

Acknowledgment

Praise be to God for guiding and helping us in every aspect of this life. Peace and mercy be upon His Holy Prophet for letting us know the importance of seeking the true knowledge.

Acknowledgment is due to King Fahd University of Petroleum and Minerals for support of this research.

I wish to thank my thesis advisor Dr. Muslim Bozyigit for his continuous guidance, help, and encouragement and committee members Dr. Subbarao Ghanta and Dr. Meng Er for their active support, help, and valuable suggestions. I also wish to thank and show my gratitude to all those who have taught me from primary school onwards.

My thanks to the graduate students of the departments of Information and Computer Science and Computer Engineering and my friends from whom I learned a lot and who made the long work hours pleasant.

Contents

List of Figures ix

List of Tables xii

Abstract(English) xiv

Abstract(Arabic) xv

1 Introduction 1

2 Scheduling: An overview 4

2.1 Introduction 4

2.2 Static Scheduling Approaches 8

2.3 Dynamic Scheduling Approaches 15

3 Parallel Architectures and Algorithms 17

3.1 Introduction 17

3.2 Models of Computation 20

3.3 Interconnection Networks for Multiprocessors: 21

3.4 Performance of Parallel Algorithms 24

3.5 Wellknown Parallel Algorithms : a literature overview 26

4 Parallel Scheduling with Precedence **31**

4.1 Introduction 31

4.2 Two processor scheduling 32

4.3 Scheduling trees in Parallel 33

4.3.1 Preliminaries 33

4.3.2 Reduction to the release-time deadline scheduling problem . . 36

4.3.3 Computing the release times and deadlines 39

4.3.4 Scheduling Tasks with Release Times and Deadlines 50

4.4 Scheduling UET task graphs in parallel 57

4.4.1 Computing the depth of a DAG 60

4.4.2 The parallel depth scheduling algorithm 64

5 Par_ETF Algorithm and its Design **67**

5.1 Problem Formulation 67

5.2 ETF heuristic: A general outlook 68

5.3 Design of the Sequential ETF heuristic 70

5.4 Objectives of Par_ETF 79

5.5 Basic Operations on the Hypercube Model 79
5.5.1 Data Broadcasting 80
5.5.2 Data Sum 83
5.5.3 Prefix Sum 84
5.6 Design of the Par_ETF 87

6 Implementation of the Par_ETF algorithm 105
6.1 Introduction 105
6.2 Par_ETF development environment 106
6.2.1 Transputer and the Helios operating system 107
6.3 CDL Programming Language 109
6.4 Realizing a hypercube using the CDL script 111
6.5 Modelling of the Par_ETF algorithm on transputers 113
6.5.1 Implementation of the Seq_ETF algorithm 115
6.5.2 Input of the Par_ETF algorithm 116
6.5.3 Par_ETF program 120

7 Performance of the Par_ETF Program 124
7.1 Incorporating the virtual time in the Par_ETF Program 124
7.2 Comparisons of Results 126
7.2.1 Test class 1: Varying the scheduling system size 127

7.2.2 Test Class 2: Varying the size of regular structured application systems . . . 133

7.2.3 Test class 3: Varying the task configuration . . . 136

7.2.4 Test class 4: Varying the size of random structured application systems . . . 139

8 Conclusion and Future work **142**

Bibliography **146**

Vita **153**

List of Figures

2.1 A typical task and system model 5

2.2 Scheduling Taxonomy 7

3.1 Approaches to concurrent programming 19

3.2 Procedure for Computing sums in parallel 27

3.3 Parallel List ranking 29

4.1 An Example of an out-tree 34

4.2 The Euler representation of the out-tree 42

4.3 Parallel algorithm for computing depths of task nodes of a tree 43

4.4 Parallel algorithm for ranking tasks 46

4.5 Parallel algorithm for computing heights of a task in a tree . 48

4.6 Computing release deadline-times : The bottom up pass . . . 54

4.7 Computing release deadline-times : The top down pass 56

4.8 Algorithm for Sequential UET scheduling 58

4.9 Parallel PRAM_MATRIX_MULT algorithm 62
4.10 Parallel algorithm for computing depths of nodes in a DAG . 63
4.11 Parallel Depth scheduling algortihm 65
5.1 The sequential ETF heuristic: Input and output 72
5.2 Broadcasting in hypercubes 80
5.3 Window broadcasting in hypercubes 82
5.4 Computing a sum in a hypercube 83
5.5 Computing the prefix sum in a hypercube 85
5.6 The Par_ETF algorithm 89
5.7 Par_ETF algorithm, Procedure Initialize 91
5.8 Par_ETF algorithm, Procedure schedule_ready_tasks 93
5.9 Par_ETF algorithm, Procedure select_ready_task 94
5.10 Par_ETF algorithm, Procedure schedule_task 94
5.11 Par_ETF algorithm, Procedure calc_new_moment 95
5.12 Par_ETF algorithm, Procedure finish_tasks 96
5.13 Par_ETF algorithm, Procedure rank_new_tasks 98
5.14 Par_ETF algorithm, calc_ready_times 100
6.1 A CDL script for a 3 dimensional hypercube 114
6.2 The input to the Seq_ETF program. The application task and system model are assumed to be same as in Figure 2.1 . 117

6.3 The output of the Seq_ETF for the input shown in Figure 2.1 118

6.4 Figure showing the input of SP(4) for the Par_ETF program 121

7.1 Scheduling completion time vs. the scheduling system size, No. of columns in scheduling system=1 129

7.2 Scheduling completion time vs. the scheduling system size, No. of columns in scheduling system=2 130

7.3 Scheduling completion time vs. the scheduling system size, No. of columns in scheduling system=4 131

7.4 Scheduling completion time vs. the no. of scheduling processors 132

7.5 Scheduling completion time vs. task graph size. The structure of task graph is hypercube like 135

7.6 Scheduling completion time vs task graph size. Task graph is random in structure 137

7.7 Scheduling completion time vs application system size 141

List of Tables

3.1 Table showing solution times for basic communication problems 25

4.1 Euler tour for the tree T defined in Figure 4.1 41

4.2 Computing the depths of the tasks 44

4.3 The Computed depths of the tasks. 44

4.4 The Release times of the tasks 44

4.5 Computing the ranks of the tasks 45

4.6 The Computed ranks of the tasks 45

4.7 The heights of the tasks defined in Figure 4.1 49

4.8 Deadlines of the tasks defined in Figure 4.1 50

5.1 The scheduling decisions made for the application in fig. 2.1 78

7.1 The execution time of Par_ETF for application system in Figure 1.1 127

7.2 Execution time of Par_ETF for a fixed application on various scheduling processor configurations 128

7.3 **The execution time of Par_ETF for various sizes of hypercube-like task graphs** . . . 134

7.4 **Description of Class 2 tests** . . . 134

7.5 **The execution time of Par_ETF for random task graphs of various sizes** . . . 136

7.6 **Description of Class 3 tests** . . . 137

7.7 **The execution time of Par_ETF for various sizes of application system arranged as hypercubes** . . . 140

7.8 **Description of Class 4 tests** . . . 141

Abstract

Name: AMIN ARSHAD ABDULGHANI

Title: A PARALLEL LIST SCHEDULING ALGORITHM: DESIGN AND PERFORMANCE

Major Field: COMPUTER SCIENCE

Date of Degree: APRIL 1993

In this thesis a parallel list scheduling algorithm for scheduling a set of n partially ordered tasks on m processors of a distributed computing system has been studied. The parallel heuristic called **Par_ETF** is based on the sequential ETF approach. The **Par_ETF** algorithm was designed on the hypercube model and implemented on a transputer environment. Logical clock was used to study the performance of the algorithm to overcome the limitations imposed by the existing parallel computing environment. The time and cost complexity of the developed algorithm is $O(n(\log n + \log m))$ and $O(mn^2(\log n + \log m))$. It has been observed through theoretical analysis and through implementation that the parallel algorithm produces the same schedules as the sequential ETF scheduling algorithm. A number of tests have been made on the **Par_ETF** program. Results obtained from these tests agree with the theoretical analysis of the **Par_ETF** algorithm.

Keywords: Parallel scheduling algorithms, hypercubes, List scheduling, transputers, distributed computing, parallel computing.

Master of Science Degree
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
April 1993

خلاصة الرسالة

اسم الطالـــب : أمين أرشد عبدالغني

عنوان الرسالة : خوارزم متوازية لعمل الجدولة : تصميم وأداء .

التخصـــص : علوم الحاسب الآلي

تاريخ الشهادة : أبريل ١٩٩٣ م

في هذه الرسالة ، تم دراسة خوارزم متوازية للقيام بجدولة مجموعة من الوظائف المرتبه جزئيًا (ن) والتى تعمل تحت نظام معالجة موزع يحتوى على (م) معالج عن طريق استخدام أسلوب جديد والمسمى (Par-ETF) ، إن الأسلوب التنقيبي المتبع (Par-ETF) صمم على مجموعة (هايبركيوب) وتم تجريبه في بيئة الترانسبيوتر . ولقد استخدمت ساعة منطقية لدراسة أداء الخوارزم .

لقد كانت درجة التعقيد الزمانية لهذا الخوارزم هي [ن.(لون + لوم)] والتكلفه هي [م.ن٢.(لون+لوم)] ، ولقد لوحظ عن طريق الحسابات النظرية والعملية أن هذا الخوارزم يعطى نفس الجداول الناتجة عن أسلوب (ETF) التعاقبى .

إن الإختبارات التى أجريت على هذا الأسلوب (Par-ETF) تتوافق مع نتائج التحليلات النظرية التى أجريت على نفس الأسلوب .

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران ، المملكة العربية السعودية

أبريل ١٩٩٣ م

Chapter 1

Introduction

During the last ten years parallelism has become truly an attractive and a viable approach to the attainment of very high computation speeds required by applications such as aircraft testing, oil exploration, bio-medical analysis and real-time speech recognition. The declining cost of computer hardware has made it possible to assemble parallel machines with very large number of processors.

A promising class of parallel computers are the distributed-memory multiprocessors. This class corresponds to loosely coupled MIMD multiprocessor systems with distributed local memories. The popular interconnect topologies include hypercube, ring, butterfly switch, hypertrees and hypernets. Message passing is the major communication method among the computing nodes. Most of these systems are scalable. Representative systems for this class of multiprocessors include iPSC, Ametek 14, NCube, BBN butterfly, CDC CyberPlus, and Warp.

One of the most challenging problems in parallel computing whose optimal solution is known to be NP- complete is the problem of 'Scheduling Parallel Programs'. The objective of the scheduling problem is to place a set of tasks that form a MIMD parallel program on a set of processors such that some desired performance measure is optimized. Schedule length and mean time spent by tasks are two examples of performance measures.

A major overhead associated with the scheduling policy is the execution time of the policy itself. It is desirable to execute the scheduling heuristic as fast as possible. In order to reduce this overhead, the objective of this thesis is the analysis and development of practical and realistic parallel heuristics for scheduling distributed applications.

We start in Chapter 2 and Chapter 3 by describing the concepts of scheduling and parallel algorithms. Chapter 2 provides an overview on the objectives of scheduling, its taxonomy and the various approaches used in the literature in solving the problem. In Chapter 3 various parallel computational, and communication models and the criteria used in evaluating the performance of those models are discussed.

In order to come up with a design of an efficient parallel heuristic for a general scheduling problem, it would be appropriate to explore various existing parallel scheduling algorithms. Chapter 4 of the thesis works on this objective by providing a detailed view of the parallel versions of 3 classical scheduling problems.

In chapter 5, the foundation of the desired parallel heuristic is laid. The algorithm is based on **ETF** heuristic [31]. Complexity results of this heuristic, referred

to as **Par_ETF** is also given in the chapter. In order to evaluate its correctness and performance the **Par_ETF** algorithm was implemented on the T-800 transputers in CDL language environment. Chapter 6 of the thesis describes the implementation details of the heuristic. The results obtained from this implementation are discussed in Chapter 7.

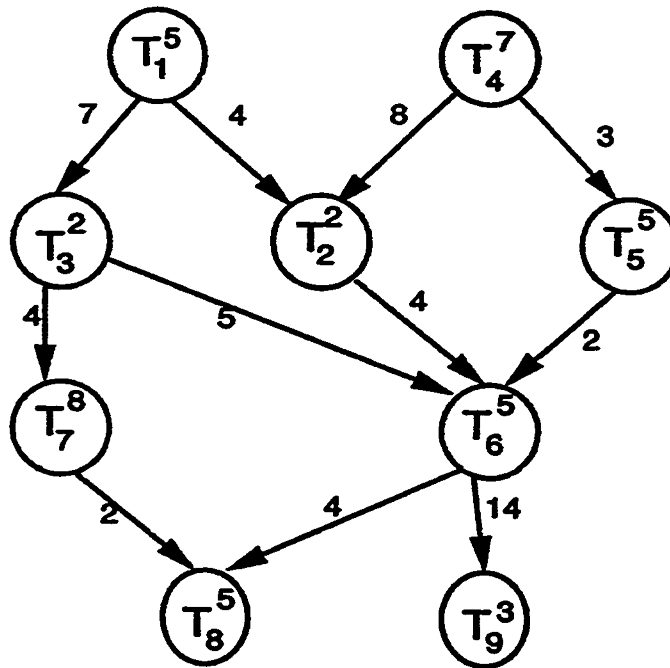
We finish our presentation in Chapter 8 by giving a conclusion to the work and then providing what in our view are possible extensions to it.

Chapter 2

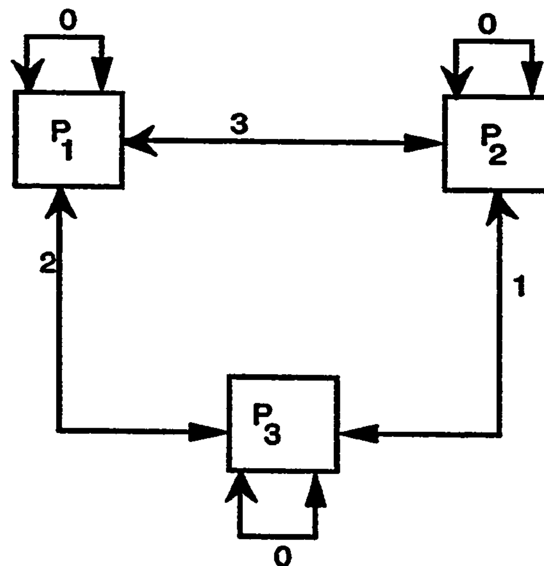
Scheduling: An overview

2.1 Introduction

Scheduling in distributed systems is the optimal placement of tasks that constitute a MIMD parallel program on a particular parallel machine in order to achieve certain objective function. Examples of such objective functions include schedule length and the mean time spent in the system. An example of a scheduling problem is shown in Fig. 2.1. Here we have two DAGs , a task graph and a system graph. A possible objective of the problem could be to map the tasks in the task graph to the processors of the system graph such that the resulting schedule length is minimal. We may classify a particular scheduling policy according to its various properties as follows:.



TASK MODEL



System model

Figure 2.1: A typical task and system model

1. **Local Versus Global :** Local scheduling is used in scheduling concurrent processes to the time slices of a single processor. Global scheduling is the assignment of tasks to processors in parallel systems.

2. **Static vs dynamic :** In static scheduling, information regarding the precedence constrained task graph is known beforehand and is fixed. Each task in the task graph has a static assignment to a particular processor, and each time that task is submitted for execution, it is assigned to that processor. The major disadvantage of static scheduling is its inadequacy in handling non-determinism in program execution. Conditional branches and loops are two program constructs that may cause non-determinism.

   In dynamic scheduling the task graph topology and labels are not known before the program executes due to branches and loops. Hence the attempt to schedule the tasks is made on the fly. The disadvantage of this kind of scheduling is its inadequacy in finding global optimums , and the overhead it incurs. This occurs because the schedule must be determined while the program is running.

3. **Adaptive Versus NonAdaptive:** A nonadaptive scheduler is a scheduler that doesn't change its behavior according to the feedback from the system. In contrast to a nonadaptive scheduler , an adaptive scheduler changes its scheduling decisions according to previous and current behavior of system. Adaptive schedulers are usually dynamic because they may collect information about the system and make scheduling decisions on the fly [49].

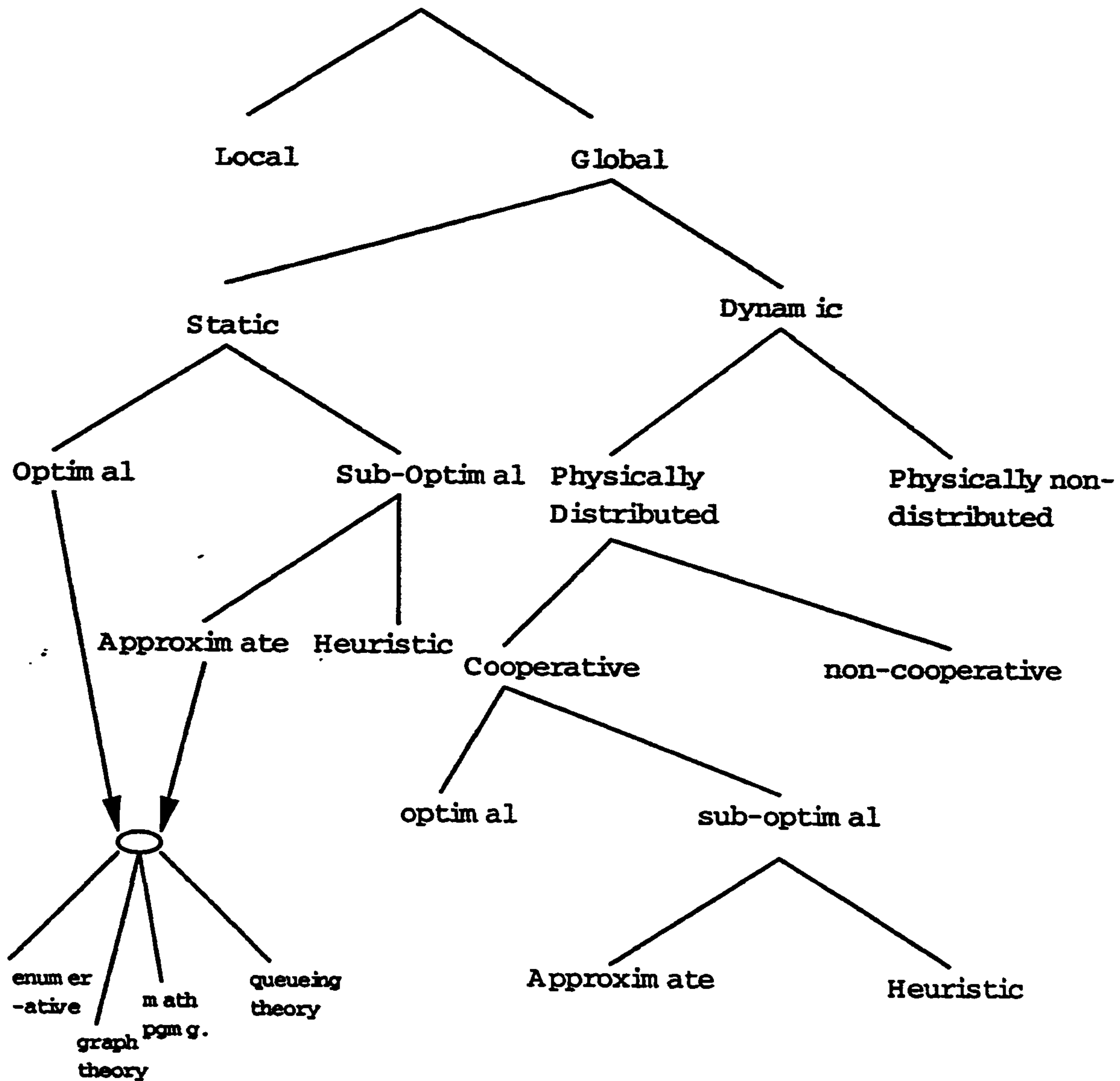


Figure 2.2: Scheduling Taxonomy

4. **Single Versus Multiple Application Systems:** In single application systems only one application can run at a time. For example, when a single application, consisting of several tasks co-operating and communicating, is run on a hypercube, the goal is to minimize the completion time of that application. In multiple application systems, several parallel applications may run at the same time in a time-sharing parallel environment. The objective here would be to minimize the response time and the average completion time per application.If the entities are jobs in the traditional batch processing sense of the term, scheduling becomes a load balancing problem where our goal is to be fair to the hardware resource of the system.

5. **NonPreemptive Versus Preemptive** With nonpreemptive scheduling, a task can not be interrupted once it has begun execution. It must be allowed to run to completion. Preemptive scheduling permits a task to be interrupted and removed form the processor under the assumption that it will eventually receive all the execution time it requires.

Static or dynamic approaches may fall into anyone of the latter 3 approaches.

2.2 Static Scheduling Approaches

The most general form of the problem is known to be NP-complete. Examples of optimization measures are minimizing total process completion time, maximizing utilization of resources in the system, or maximizing system throughput.

When optimal solutions are computationally infeasible suboptimal solutions may be tried. Suboptimal solutions may be labeled as **Approximate** or **heuristic.**

Suboptimal-approximate solutions use the same formal computational model for the algorithm but instead of searching the entire solution space for an optimal solution, we are satisfied when we find a "good" one. The factors which determine whether this approach is worthy of pursuit include:.

1. The time required to evaluate a solution.
2. The ability to judge according to some metric the value of a solution.
3. Availability of a mechanism for effectively pruning the solution space.

Static optimal or suboptimal-approximate solutions [5], [14], [30], [51] can be divided into four basic categories of task allocation algorithms:.

1. Solution space enumeration and search
2. Graph Theoretic
3. Mathematical programming
4. Queuing theoretic

In [51] an optimal enumeration approach is used for task assignment problem. The criterion function is defined in terms of optimizing the amount of time a task will require for all intertask communication and execution, where the tasks submitted

by users are assumed to be broken into suitable modules before execution. The cost function is called a minimax criterion since it is intended to minimize the maximum execution and communication time required by any single processor involved in the assignment. Graphs are then used to represent the task to processor assignments and the assignments are then transformed to a type of graph matching known as weak homomorphisms. The optimal search of this solution space can be done using the A* branch and bound algorithm.

The model given in [42] represents an example of an optimum integer programming formulation employing a branch and bound technique to search the solution space. The model described does not consider the effects of precedence relations in the data flow subtasks.

Nonlinear programming has also been used for task allocation under an assumption that the given task can be split into arbitrary size subtasks [1]. A limitation of such research is that it is not directly applicable to practical situations where partitioning can usually be done only at specific points.

In [11] a static optimal queuing theoretical solution is used to minimize the execution time of the entire application, and the algorithm is derived from results in Markov decision theory.

Sinclair, J.B. , presented an algorithm in [52] named as minimum independent assignment costs underestimate (MIACU) to find an optimal assignment of distributed tasks. This is a branch and bound with underestimates algorithm and simulation showed that it gives excellent results in reducing the average time and

space complexities for finding optimal assignments.

In [30], Hu gave an optimal linear scheduling algorithm for cases in which the task graph is a tree and the execution time for each task is one time unit. The algorithm he devised belongs to the more general class of **height priority** algorithms which have also been called CP (critical path), LP (longest path) , or LPT (longest process time) algorithms.

Coffman and Graham [14] gave an optimal length scheduling algorithm similar to Hu's algorithm to schedule for an arbitrary graph containing unit-time delay tasks on a two-processor system. The algorithm they introduced had a running time complexity of $O(n^2)$ where n is the number of tasks.

Stone has efficiently solved the problem of allocating a number of tasks to two processors by performing a polynomial min-cut algorithm on a graph [55]. The processes are represented as a set of nodes in a graph and the inter-task communication costs are represented by the weights on the edges of the graph. The objective function is the sum of the execution and communication costs. In order to represent the execution costs of a task with respect to each processor, two more nodes, say P1 and P2 are added to the graph, corresponding to the two processors. The weight on the edge connecting a task node to P2 is the execution cost of the corresponding task on processor P1, and the weight on the edge connecting P1 to a task node is the execution cost of that task on processor P2. No resource constraints are imposed on any of the processors and, in general, any process is free to reside on any processor while pre-allocation of certain tasks on specific processors can be

accommodated by the method. Then, by applying a max-flow min-cut algorithm, the processes are optimally partitioned into two sets, each allocated to a particular processor. The minimum weight cutset obtained from this solution determines the module assignment which is optimal in terms of total cost.

Bokhari in [6] has proposed optimal solutions to the following problems:

1) partition chain-structured parallel or pipelined applications over chain-connected systems;

2) partition multiple chain-structured parallel or pipelined applications over single-host multiple-satellite systems.

3) partition multiple arbitrarily structured serial programs over single-host multiple-satellite systems.

4) partition single-tree structured parallel programs over single-host multiple identical satellites systems.

Bokhari solves problem 1 by a minimum bottleneck path algorithm. For problems 2 to 4 he uses an algorithm for a novel graph theoretic problem: **the minimum sum bottleneck path problem.**

Simulated Annealing approach has also been applied as heuristics in solving scheduling problems [40], [46]. This is a stochastic strategy for selecting a minimal or maximal configuration of the states of a system. It differs from conventional iterative improvement methods that it always converge asymptotically to the global

minimum [41]. Conventional iterative methods begin with an initial feasible solution, repeatedly consider changes in the current configuration, and accept only those that improve the objective function. However the annealing method probabilistically accepts configurations which temporarily deteriorate the quality of the system. The acceptance probability is a function of the change in the objective function, and a temperature parameter θ. As the parameter is slowly reduced, fewer non-improving moves are accepted. Thus a coarse global search evolves into a fine local search for optimality, and the probabilistic 'jumps' provide an escape from non-global optima.

Heuristic methods are another way to obtain suboptimal solutions [19], [47], [38]. Using intuition, we try to come up with heuristics that make use of special parameters that affect the system in an indirect way. For example, tasks that heavily communicate with each other should be placed on the same processor or on processors that are close to each other. This might avoid additional communication delay between processors.

Chu and Lan in [12] propose a heuristic solution to task assignment problem consisting of two phases. In the first phase n tasks are reduced to $G <= m$ groups. Each group generated at the end of the first phase is a set of tasks which will be assigned as a single unit to a processor. In phase two an exhaustive search is performed through the new assignment tree to yield a minimum-bottleneck assignment. The heuristic considers the effect of precedence relationships, communication, and execution times when evaluating the task allocation.

Kaufman in [34] reported an algorithm similar to Hu's algorithm that works on a

tree containing tasks with arbitrary execution time. This algorithm finds a schedule in time bounded by $(1+(m-1)a_{\mathrm{long}})/a_{total}$, where m is the number of processors , a_{long} is the longest execution time, and a_{total} is the summation of all task execution times.

One of the classes of scheduling heuristics is list scheduling. In list scheduling each task is assigned a priority. Whenever a processor is available, a task with the highest priority is selected from the list and assigned to a processor. Schedulers in this class differ only in the way that each scheduler assigns priorities to nodes. Priority assignment results in different schedules because tasks are selected in different order.

In [47] schedules are computed by a modified list scheduling technique. The heuristic called the MH heuristic uses the level of each node in the task graph as each node's priority. It breaks ties by selecting the task with the largest number of immediate successors. If this does not break the tie, it selects a task at random.

When a task is ready, a processor is selected to run that task in such a way that the task cannot finish earlier on any other processor. The parameters T and C, which reflect the speed of the processors, interconnection topology, and contention are considered when a processor is selected. The selected task is then allocated to the selected processing element.

When a task is done, the status of the immediate successors of the finished task is modified. So when a task finishes execution, the number of conditions that prevent any of its immediate successors from being run is decreased by one. When

the number of conditions associated with a particular processor becomes zero, then that successor node can be scheduled.

Hwang, Chow, Anger, and Lee in [31] presented a new heuristic named **Earliest Task First (ETF)** which effectively reduces communication delay and also maintains the strength of the classical list scheduling. The approach used by ETF is greedy and the makespan ω_{ETF} generated by ETF always satisfies $\omega_{\mathrm{ETF}} <= (2 - 1/n)\omega^{(i)}_{\mathrm{OPT}} + C$, where $w^{(i)}_{\mathrm{OPT}}$ is the optimal makespan without considering communication delay and C is the communication requirements over an immediate predecessor-immediate successor pairs along one chain. The running time complexity of the heuristic is shown to be $O(mn^2)$ where m and n are the number of processors and the number of tasks respectively. Further details of this algorithm are discussed in chapter 5, where we are interested in parallelizing it.

2.3 Dynamic Scheduling Approaches

In dynamic scheduling if the work involved in making decisions for scheduling is distributed among different processors, then the scheduler is physically distributed. Otherwise, if the scheduling responsibility is assigned to only one processor, it is called **physically non distributed**. An example of a physically nondistributed scheduler is **Medusa** [44]. In this system, the functions of the operating system (eg file system, scheduler) are physically partitioned and placed at different places in the system. Hence, the scheduling function is placed at a particular place and is accessed by all users at that location.

Physically distributed schedulers are further classified as either **Cooperative or noncooperative.** In cooperative schedulers local schedulers at each processor cooperate and work together to come up with a global schedule that is based on the situation in the whole system. In non-cooperative systems individual processors work independently and arrive at decisions that will affect local performance only [35].

Beneath the cooperative dynamic scheduling branch of the taxonomy tree, we distinguish between optimal and suboptimal solution cases. An example of a physically distributed, cooperative, suboptimal, heuristic solution is given in [50]. This is a heuristic derived from the area of Bayesian decision theory. The algorithm uses no *a priori* knowledge regarding the task characteristics, and is dynamic in the sense that the probability distributions which allow maximizing decisions to be made based on the most likely current state of nature are updated dynamically. Monitor nodes make observations every p seconds and update probabilities. Every d seconds the scheduler itself is invoked to approximate the current state of nature and make the appropriate maximizing action. It was found that the parameters p and d could be tuned to obtain maximum performance for a minimum cost. Since we are not concerned with dynamic scheduling in our work, we will not discuss this subject further.

Chapter 3

Parallel Architectures and Algorithms

3.1 Introduction

In this section we would study the issues and concepts of parallel computations and parallel architectures. This would help us in designing and evaluating parallel scheduling algorithms. A parallel algorithm is a solution method for a given problem destined to be performed on a parallel computer. The design of a parallel algorithm is affected by the computational of the underlying parallel computer.

To implement parallel algorithms, three main approaches have been suggested [32]:

1. **The Compiler approach:** In this approach one writes a program in a conventional sequential language and lets an intelligent compiler detect the opportunities for parallelism that are in the program. For example, vectorizing compilers have been used for restructuring innermost loops of programs to be executed on vector processors [3]. Parallelizing compilers try to extract parallelism at instruction and/or loop level from a sequential program, thus making explicit parallel programming unnecessary. However, a compiler can enhance the performance by only a limited factor, due to the difficulty in detecting parallelism in a complex program mix.

2. **Language Extensions:** In this approach, sequential languages are extended with architecture-oriented constructs to support concurrent programming. Usually, only one type of parallelism is supported in each extended language. For instance, concurrent C language has been extended from C for the Flex/32 multicomputer [20]. These extensions are machine-oriented and thus users may have to re-code the parallel algorithm in another extended language, when the target machines are changed.

3. **New Languages:** With this approach, new concurrent languages have been developed for supporting parallel processing. Quite a few concurrent languages have been proposed in recent years, including Concurrent Pascal, Modula-2, Occam, concurrent C and SISAL [32]. Unlike an extended language, these new languages contain some application-oriented parallel constructs. However, each of these languages is usually designed to support one form of parallelism.

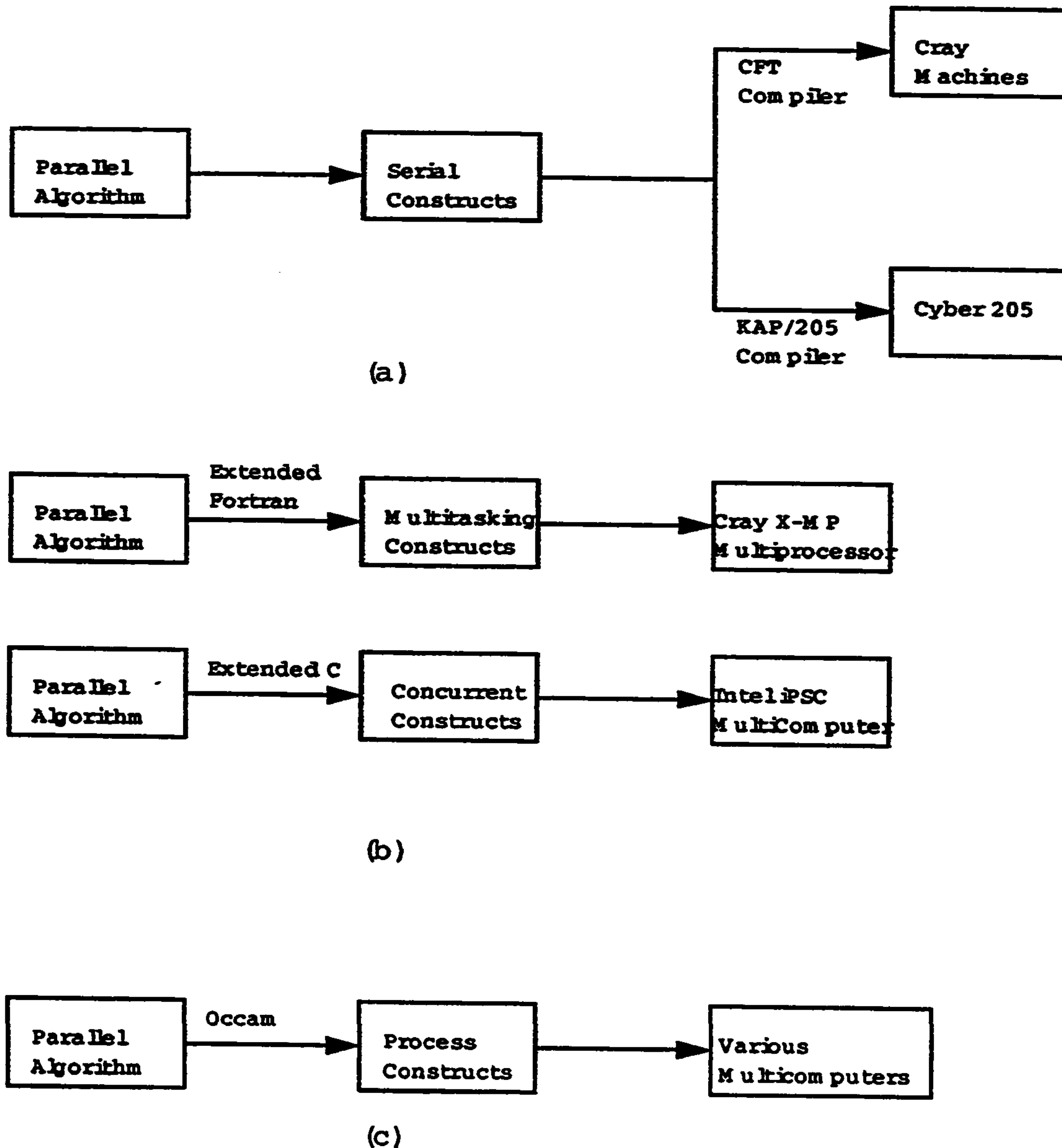


Fig: 3.1 Three approaches to concurrent programming a) Smart Compiler Approach b) Language Extension approach c) New Language approach

Figure 3.1: Approaches to concurrent programming

3.2 Models of Computation

A generally accepted taxonomy for multiprocessor architectures has yet to be developed. Flynn in [21] classifies computers into four categories in terms of the sequence of instructions performed by the machine and sequence of data manipulated by the instruction streams.

1) Single Instruction Single Data Computers (SISD): SISD computers consists of a single processing unit receiving a single stream of instructions that operate on a single stream of data at each step.

2) Multiple Instruction, Single Data Computers (MISD): In MISD computers, there are N streams of instructions and one stream of data. At each step, one datum received from memory is operated upon by all the processors each according to the instruction it receives from its control unit.

3) Single Instruction Multiple Data Computers (SIMD): In SIMD computers, all processors execute the same instruction, each on a different datum. All the processors operate synchronously.

4) Multiple Instruction Multiple Data Computers(MIMD): This is the most general and most powerful class of computers. Here, there are N processors, N streams of instructions, and N streams of data. The processors can potentially execute different programs on different data while solving different subproblems of a single problem.

3.3 Interconnection Networks for Multiprocessors:

The processors and memories in a multiprocessor can be connected in several ways. Based on the interconnection used, multiprocessors may be classified as shared-memory multiprocessors and distributed multiprocessors.

1) Shared Memory (SM) Computers:

This class is also known in the literature as the Parallel Random Access Machine (PRAM) Model. The N processors share a common memory. When two processors wish to communicate, they do so through the shared memory. The class of shared memory computers can be further divided into four subclasses according to whether to two or more processors can gain access to same memory location simultaneously.

1. **Exclusive-Read, Exclusive Write (EREW) SM Computers:** No two processors are allowed simultaneously to read from or write into the same memory location.
2. **Concurrent-Read, Exclusive-Write (CREW) SM Computers:** Multiple processors are allowed to read from the same memory location, but the right to write is still exclusive.
3. **Exclusive-Read , Concurrent Write(ERCW) SM Computers:** Multiple processors are allowed to write into the same memory location but read accesses remain exclusive.

4. **Concurrent-Read, Concurrent -Write(CRCW) SM Computers:** Both multiple-read and multiple-write privileges are granted.

Even though EREW is the weakest of the four subclasses of the shared memory approach, we can through simulation allow multiple accesses at the cost of either increasing the space and/or time requirements of an algorithm.

2) Distributed Computers

The shared memory model can play a usefule role as a theoretical support for measuring the limits of parallel computations. However, it is not easy to realize this model physically because of physical fan-in limitaitons. In a physically realizable assemblage, we can only expect any computing element to have a small number of external connections. We must therefore consider parallel assemblage in which a large number of communication processors, each with its own memory, are connected together, but where each processor communicates with a small number of other p processors.

A N-processor fixed interconnection network may be viewed as an undirected graph, where vertices correspond to processors, and edges correspond to communication links. Each processor P, $0 \leq i \leq N-1$, has a large local memory. There is no shared memory. We assume that the processors operate synchronously and they communicate with one another by sending and receiving data packets over the communication links provided by the network.

Topologies are evaluated by their diameter (maximum distance between any pair

of nodes), connectivity (minimum number of nodes that have to be deleted for the network to become disconnected), communication protocols, flexibility provided in running a broad variety of algorithms and the communication delay they incur for frequently used tasks. The frequently used tasks in parallel computations include single and multiple node broadcast, single and multinode accumulation, single node scatter, single node gather, and total exchange. The most popular of these networks are as follows:

i) **Linear Array:** Here we interconnect N processors in the form of a one-dimensional array. A processor P_i is linked to its neighbors P_{i-1} and P_{i+1}. through a two-way communication.

ii) **Two-Dimensional Array or mesh:** A mesh is obtained by arranging the N processors into an m*m array, where $m = N^{1/2}$. Clearly, the diameter of such a network is 2m. The processor in row j and column k is denoted by P(j,k) where $0 \leq j \leq m-1$ and $0 \leq k \leq m-1$. A two-way communication line links P(j,k) to its neighbors P(j+1,k), P(j-1,k), P(j,k+1), and P(j,k-1). Processors on the boundary rows and columns have fewer connections.

iii) **Tree Connection:** In this network, the processors form a complete binary tree. Such a tree has d levels numbered 0 to d-1, and $N = 2^d - 1$ nodes each of which is a processor. Each processor at level i, is connected by a two-way line to its parent at level i+1 and to its two children at level i-1. The root processor has no parent, and the leaves have no children. A tree network with N processors provides communication between every pair of processors with a minimum number of links (N-1). One disadvantage of a tree is its

low connectivity. The failure of any one of its links creates two subsets of processors that cannot communicate with each other.

iv)Hypercube: Assume that $N = 2^d$ for some $d \geq 1$ processors are available. A hypercube or d-cube is obtained by connecting each processor to d neighbors. The d neighbors P_j of P_i are defined as follows: The binary representation of j is obtained from that of i by complementing a single bit. We can map a linear array of 2^d nodes into a hypercube. This amounts to constructing a sequence of 2^d distinct binary numbers with d bits each, with the property that successive numbers in the sequence differ in only bit. Such sequences are called Gray code. A particular type of Gray code called reflected Gray Code has a property that first and last number also differ in only one bit. Thus it provides a mapping of a ring with 2^d nodes. Similarly, a d-cube can be arranged as a two-dimensional mesh and a two-rooted balanced tree. In a d-cube there are exactly d independent paths between any pair of nodes. Since none of the paths share any node except the two end nodes, the node connectivity of the d-cube is d. Of the d independent paths k have k links and the remaining d-k have k+2 links, where k is the number of bits the identity numbers of the two end nodes differ by.

3.4 Performance of Parallel Algorithms

Let $t_1(n)$ be the running time of an optimal sequential algorithm for solving a problem Π where n is the length of the input of Π. Let $t_p(n)$ be the running time

Problem	Ring	Tree	Mesh	Hypercube
Single node broadcast (or single node accumulation)	$\theta(p)$	$\theta(\log p)$	$\theta(p^{1/2})$	$\theta(\log p)$
Single node scatter (or single node gather)	$\theta(p)$	$\theta(p)$	$\theta(p)$	$\theta(p/\log p)$
Multinode broadcast (or multinode accumulation)	$\theta(p)$	$\theta(p)$	$\theta(p)$	$\theta(p/\log p)$
Total exchange	$\theta(p^2)$	$\theta(p^2)$	$\theta(p^{3/2})$	$\theta(p)$

Table 3.1: Table showing solution times for basic communication problems

of a parallel algorithm for solving Π with p processors. Then the cost of a parallel algorithm is defined as its running time multiplied by the number of processors it requires in order to execute i.e. $c(n) = t_p(n) * p(n)$. A parallel algorithm, for a given problem, is said to be cost optimal if its cost complexity is equal to the time complexity of the best available sequential solution [2].

The speed-up $S_p(n)$, of the parallel algorithm for solving Π is $t_1(n)/t_p(n)$. It measures how many times a parallel algorithm is faster than a sequential one. Clearly, $S_p(n) \leq p$.

The efficiency, $E_p(n)$ of the parallel algorithm is $S_p(n)/p$. It measures how effective each processor in a parallel algorithm is relative to a sequential algorithm. It normalizes the speedup. A parallel algorithm that runs in time $t_p(n)$ is said to be efficient if $E_p(N) = \theta(1)$ and is said to be almost efficient if $E_p(N) = \Omega(1/\log n)$. It should be noted that an efficient algorithm may not necessarily be fast. Our primary goal would be to design efficient or almost efficient algorithms that also run as fast as possible.

The efficiency of a parallel algorithm is measured by its running time and the number of processor it uses. These two measures are strongly related. The following theorem due to Brent [7] implies that we can always slow down a parallel algoritthm by reducing the number of its processors with the same processor-time product. This is the reason why we often measure the efficiency of a parallel algorithm by its minimal running time and the number of processors required to achieve this running time.

Theorem 3.4.1 *A PRAM algorithm requiring t parallel steps and a total of x operations can be implemented by a p-processor PRAM within $\lceil x/p \rceil + t$ parallel steps.*

Proof of Theorem 3.4.1 *Let x_i be the number of operations performed in step i, $1 \leq i \leq t$. The p-processor PRAM can perform the x_i operations in $\lceil x_i/p \rceil$ steps. Hence the total number of steps on the p-processor PRAM is*

$$\Sigma_{i=1}^{t} \lceil x_i/p \rceil \leq \Sigma_{i=1}^{t} (\lceil x_i/p \rceil + 1) \leq \lceil x/p \rceil + t$$

□

3.5 Wellknown Parallel Algorithms : a literature overview

In this section, our purpose is to review the available literature on a number of the parallel algorithms that would be encountered in this thesis. For the sake of

```
Procedure PARALLEL_SUM
begin

    for j=1 to n dopar
            S[0,j] := s_j;
    odpar
    for h=1 to log N do
            for j=1 to 2^((logN)-h) do par
                    S[h,j] := S[h-1, 2j-1] + S[h-1, 2];
            odpar
    endfor

end
```

Figure 3.2: **Procedure for Computing sums in parallel**

simplicity, these algorithms are assumed to be based on the EREW PRAM model.

We start with the problem of computing the sum of n numbers, $S_1, S_2, \ldots, S_n$, where n is a power of 2. We assume the existence of a balanced binary tree T with n leaves. The nodes of the tree are denoted by [h,j], where h is its height in the tree and j its position within the other nodes at the same height. The height of all the leave nodes is 0. Thus, node [0,j] represents leaf j and corresponds to the element s_j. Let us associate an element $S[h, j]$ with each node $[h, j]$ of the tree. For each internal node $[h, j]$ represents the sum of all the elements in the leaves of the complete subtree rooted at node $[h, j]$. The algorithm is given in Fig. 3.2.

It is easy to see that the required sum $S[\log n, 1]$ can be computed in $O(\log n)$ time using n, processors. Now, if we apply **Brent's theorem** then we get an alternative implementation that uses $p = n/\log n$ processors and runs in $O(\lceil n/p \rceil + \log n)$ time, i.e. $O(\log n)$ time. This algortihm can be adapted to compute other operations such as the maximum of a list, and broadcasting a value of a memory cell.

Another problem encountered in this thesis is the **list-ranking problem**. Here we have a linked list L of n nodes whose order is specified by an array S such that $S[i]$ contains a pointer to the node following node i on L, for $1 \leq i \leq n$, and our objective is to determine the distance $R[i]$, of each node i from the end of the list. It is assumed that when i is the end of the list $S[i] = 0$. The problem can be solved in parallel by using the algorithm shown in Fig. 3.3, which is based on the pointerjumping technique The running time of the above algorithm is $O(\log n)$ and the cost is $O(n \log n)$. This algorithm is non-optimal in view of the the linear time sequential algorithm.

An optimal method for list ranking is explained in [33, pp. 92-108]. The strategy involves partitioning the input list into approximately $n \log n$ blocks $\{B_i\}$, each containing $O(\log n)$ nodes, ranking each node within its block by using an optimal sequential algorithm, and combining these preliminary ranks using an $O(\log n)$ time parallel algorithm. The running time of the algorithm remains the same i.e. $O(\log n)$ but the cost is reduced to $O(n)$.

The other procedures that are of interest to us is parallel merging and sorting.

```
Procedure list_Ranking
begin
    for i=1 to n dopar
        if (S[i] ≠ 0)
            R[i] := 1;
        else
            R [i] := 0;
        fi
    odpar
    for i:=1 to n dopar
        Q[i] := S[i];
        while (Q[i] ≠ O and Q[Q[i]] ≠ 0) do
            R[i] := R[i] + R[Q[i]];
            Q[i] := [Q[i]];
        endwhile
    odpar
end.
```

Figure 3.3: **Parallel List ranking**

Hagerup et al. have described a simple and optimal EREW PRAM algorithm for the task of merging two sorted sequences of total length n [26]. The running time of the algorithm is $O(\log n)$ using $O(n/\log n)$ processors. Since, the lower bound of merging two sorted sequences of total length n in an EREW PRAM is $\Omega(\log n)$ [54], the algorithm is not only optimal in its cost but also in its running time.

As for the sorting algorithm, it can be derived from the above merging algorithm by using the binary tree method. Consider a n-leaf binary tree in which the inputs to the binary tree are distributed one per leaf. The task at each internal node U of the tree, is to compute the sorted order for the items initially at the leaves of the subtree rooted at U. The computation proceeds up the tree, level by level, from the leaves to the root, as follows. At each node we compute the merge of the sorted sets computed at its children. Thus after $\log n$ iterations we will have the sorted set of n numbers at the root. The running time of this algorithm is $O(\log^2 n)$ using $O(n/\log n)$ processors.

Cole has given a faster sorting algorithms which runs in $O(\log n)$ time. The algorithm utilizes the observation that the merges at the different levels of the tree can be pipelined [15]. The cost of the algorithm remains $O(n/\log n)$.

Chapter 4

Parallel Scheduling with Precedence

4.1 Introduction

In this chapter we are going to discuss basic parallel list scheduling algorithms for restricted task and system models. The algorithms explored are the parallel two processor scheduling, parallel scheduling of trees, and scheduling of UET task graphs. In all of these problems the application is modeled as a graph $G = (\Gamma, E)$ where $\Gamma = \{T_1, T_2, \ldots, T_n\}$ is a set of tasks with execution times and E is the set of edges representing the precedence relationships between the tasks. A task will not be able to execute unless all its predecessors have been executed. The system is assumed to contain m processors. The schedules produced in the first two cases

are optimal while in the last case the schedules produced though not optimal, are bounded by a known performance bound. The exploration of these algorithms would give us the insight needed to develop a parallel heuristic algorithm for a general scheduling problem. This is because scheduling algorithms are mostly sequential in nature, and coming up with a parallel version is a very challenging theoretical and practical problem.

4.2 Two processor scheduling

The two processor scheduling problem is one of the classical problems in scheduling. This problem has a long history and rich literature. The first sequential polynomial time solution for the problem had a complexity of $O(n^4)$ [22] . Three years later Coffman and Graham published an $O(n^2)$ algorithm [13]. Gabow [23] found an algorithm that, when combined with Tarjan's union find result [24] runs in O(n+e) time and hence is asymptotically optimal.

The first parallel algorithm for the problem was given by Vazironi et al. [56]. Their algorithm is a randomized parallel algorithm and the expected running time of the algorithm is a polynomial in the logarithm of the number of tasks. The first deterministic two-processor scheduling algorithm was proposed by Helmbold and Mayr [27]. The algorithm is rather complex and consists of three main steps. The first step computes the length of an optimal schedule. Next the algorithm locates the empty slots or holes in *lexicographically maximum jump sequence*. In the final step of the algorithm the tasks are assigned to jumps in the jump sequence.

The requirements for the entire algorithm are $O(n^7L^3)$ scheduling processors and $O(\log n \log L)$ time, where L is the depth of the task graph.

4.3 Scheduling trees in Parallel

4.3.1 Preliminaries

In this section, our objective is to construct a height priority schedule in parallel in polylog time for a set of n unit-length tasks on m processors. The algorithm presented is an integral algorithm which includes all the steps of scheduling based on mostly existing methods.

The precedence graph for the tasks is an outforest $G = T = (\Gamma, E)$ in which every task $T_i \in \Gamma$ has at most one incoming edge. An example of an out-tree which is a forest having a unique root is shown in Figure 4.1.

To obtain an optimal schedule for an inforest, in which every task has at most one outgoing edge, we reverse the precedence graph to an outforest and apply our algorithm [18].

Definition 4.3.1 *Define a* **profile** μ *to be a function from N into N, where* $\mu(i)$ *is the number of processors available at the* i^{th} *time slot (the interval [i,i+1]). If a profile has only one value m, then it is called straight. Here we would be dealing with only straight profiles.*

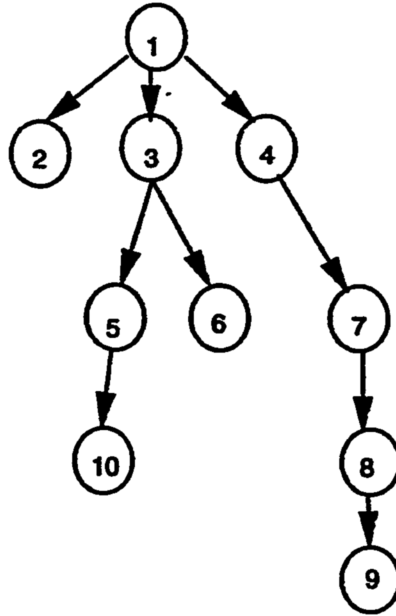


Figure 4.1: An Example of an out-tree

Definition 4.3.2 *Define a* **schedule** *s for the tree T and a profile μ to be a function from the set of tasks of the tree T onto an initial segment $0, \ldots l-1$ of N such that:*

1) $s^{-1}(\tau) \leq \mu(\tau)$ *for all* $\tau \in \{0, \ldots l-1\}$,

2) If T_j is a successor of T_i in T, then $s(T_i) < s(T_j)$

Definition 4.3.3 *A schedule s is* **greedy** *if at every slot maximum number of tasks are scheduled, i.e. $|s^{-1}(k)| < m$ implies that every task T_j s.t. $s(T_j) > k$ is a successor of some task T_i in $s^{-1}(k)$.*

Note the definition of a greedy scheduling algorithm fits our description of the list scheduling algorithm.

Definition 4.3.4 *Define the* **height** h *of task* T_i *to be the length of the longest path starting with* T_i. *A schedule* s *is a* **height priority-schedule** *if tasks of higher height are preferred over tasks of lower height. A height-priority schedule has a property that if:* $s(T_i) > s(T_j)$ *and* $h(T_i) > h(T_j)$ *then* T_i *is a successor of some task* z *with* $s(T_k) = s(T_j)$.

Height priority schedules are optimal for straight profiles in the case where the graph is an in-forest or an out-forest [30]. It is easy to find a height priority schedule in $O(n \log n)$ sequential time: Fill the slots in increasing order; keep track of the set of tasks of depth zero, i.e. the tasks that are candidates to be scheduled in the next slot; pick tasks according to highest height using a priority queue.

There are even linear time algorithms for finding a height priority schedule [8]. Brucker et al. describes, a linear time algorithm for finding a height priority schedule for in-forests. Their algorithm assumes that the task graph is given in the form of an in-tree and the tasks are given in a priority list L which has a property that all predecessors of a task T_i occur before T_i in L. The method uses a number of auxiliary variables. For each T_i , $1 \leq i \leq n$, the variable $R[T_i]$ is one greater than the latest starting time assigned so far to any immediate predecessor of T_i. For each integer time t, $0 \leq t \leq (n-1)$, the variable $N[t]$ shows the number of tasks assigned to start at time t. The variable FIRST gives the least value of t for which $N[t] < m$. Initially, all these auxiliary variables are set to 0. Then for each value of i, increasing from 1 upto n, it determines $s[T_i]$ and updates the auxiliary variables as follows:

1. $t := \max\{R[T_i], \text{FIRST}\}$;

2. Set $s(T_i) := t$;

3. Set $N[t] := N[t] + 1$;
   If $(N[t] = m)$ then FIRST:=t+1;

4. For the immediate successor T_j of T_i, set: $R[T_j] := max\{R[T_j], t + 1\}$;

One can also design a linear algorithm for an outforest using the notion of Elite and Median introduced in [17].

To be able to construct a height priority schedule in parallel we need to be able to predict the remaining graph at various times. This at first sight seems difficult to do in polylog time because of the seemingly sequential nature of the precedence constraints. We overcome this difficulty by assigning with every task an integer release time and deadlines and then dropping the precedence constraints. The release times and deadlines are chosen such that the obtained schedule does not violate the precedence constraints.

4.3.2 Reduction to the release-time deadline scheduling problem

Define the release time, $r(T_i)$, of a task T_i as the earliest time at which T_i can start and the deadline, $d(T_i)$, of T_i as the time by which T_i should complete. A unit-length scheduling problem with release times and deadlines is given by a set Γ of n-unit length tasks with each task T_i having a non-negative integer

release time $r(T_i)$ and a positive integer deadline $d(T_i)$. Tasks have to be scheduled on m processors, s.t. every task is executed during its release time deadline interval.

Definition 4.3.5 *Define a* **release-deadline schedule** $\hat{s}$ **for** Γ *to be a function that maps the set of tasks* Γ *into* N *such that:*

1. $r(T_i) \leq \hat{s}(T_i) \leq d(T_i) - 1 \; \forall T_i \in \Gamma$ *and*
2. $|\hat{s}^{-1}(k)| \leq m$, *for* k *in* N.

A schedule $\hat{s}$ for Γ can sequentially be obtained as follows: Scan the slots in increasing order and among all the unscheduled tasks that were released at the current slot or before, schedule the task with with the earliest deadline. Each slot is assigned as many slots as possible. The produced schedule is called **ED-schedule (Earliest Deadline-schedule)**. An ED-schedule has a following property:

For any two tasks T_i and T_j and a time slot $k \geq 1$:

1) if $\hat{s}(T_i) = k$ and $|\hat{s}^{-1}(k-1)| < m$ then $r(T_i) = k$;

2) $\hat{s}(T_i) < \hat{s}(T_j)$ and $d(T_j) < d(T_i)$ implies $r(T_j) > r(T_i)$.

Our aim is to assign for each task in the outforest T_i a release time and a distinct deadline such that the corresponding unique ED-schedule is a height priority schedule for the outforest and in this schedule precedence constraints are not violated. We define the **release time** for a task T_i to be equal to its depth $\pi(T_i)$ in the outforest

i.e. $r(T_i) = \pi(T_i)$. To compute the deadline $d(T_i)$ of a task T_i, we construct a list L_d of the tasks in Γ, sorted according to nonincreasing height where vertices of the same height are sorted according to their Eulerian path numbering. The deadline of the task T_i is defined to be as follows:

$d(T_i) = n +$"the index of T_i in list L"

Note that the deadlines are large enough that there always exists a schedule that doesn't violate release times and deadlines.

From the above definitions, we see that the deadlines of the tasks have the following property

i) If $h(T_i) > h(T_j)$ for some tasks T_i and T_j, then $d(T_i) < d(T_j)$.

ii) For any tasks $T_i, T_{i'}, T_j, T_{j'}$ s.t. $T_{i'}$ is an immediate successor of T_i of height $h(T_i) - 1$ and $T_{j'}$ is the immediate successor of T_j of height $h(T_j) - 1$

$$d(T_i) < d(T_j) \Rightarrow d(T_{i'}) < d(T_{j'}).$$

It has been shown in [18] that for any outforest :

1. The ED-schedule of the desired release-time deadline problem does not violate the precedence constraints of the outforest T.
2. The derived ED-schedule is a height priority and hence an optimal schedule for the outforest T.

4.3.3 Computing the release times and deadlines

In this subsection we will show how the release times and the deadlines are computed on an *EREW PRAM* using an Eulerian path of the given outforest. The input to our problem consists of m, the number of processors available at any time slot, and E, the list of nontransitive edges that define the outforest. Denote an edge from T_i to T_j by $< T_i, T_j >$ where T_i is the parent of T_j or symbolically T_i=$p(T_j)$.

For the sake of simplicity we assume that the given outforest consists of only one tree. If this is not the case, we can add a dummy root r such that the outforest on $(n-1)$ tasks becomes an outtree of n tasks by first identifying the roots of the outforest by examining the in-degrees of each task in the forest. We then broadcast the dummy root r to each of the identified root nodes. Finally, we add the extra edges, from the dummy root r to the identified roots in the outforest, to our original list of edges. All of these steps can be performed on EREW PRAM model in $O(\log n)$ using $O(n/\log n)$ processors.

Our first step in computing the release times and deadlines for the outtree T is to construct a Eulerian path Ψ, of the outtree. We define a **Eulerian graph** G to be a directed graph s.t. for every node $v \in G$, indegree(v) =outdegree(v). A path that traverses each and every edge in this graph exactly once is referred to as the **Eulerian path.**

If the Eulerian graph G is represented by an adjacency list L of the vertices then we can obtain a corresponding Eulerian path for G by specifying a **successor function** σ mapping each edge e in G into the arc $\sigma(e)$ in G, that follows e on

the path. For a vertex v of degree d and an adjacency list L, where $L[v] =< u_0, u_1, \ldots, u_{d-1} >$, $\sigma(u_i, v)$ is defined as follows:.

$$\sigma(< u_i, v >) =< v, u_{i+1} \bmod d > \text{ for } 0 \leq i \leq (d-1).$$

In order to obtain the successor of an edge in G in O(1) operation we need to make the adjacency list circular and let each edge $< u_i, v >$ point to its reversal edge $< v, u_i >$.

Coming back to our problem we note that :

1. Our tree is not necessarily a Eulerian graph, and
2. It is not in the required adjacency representation.

To put the tree in the required Eulerian representation, we create for every original directed edge $< T_i, T_j >$, its reversal $< T_j, T_i >$. The set of edges directed away from the root are referred to as the **forward** edges while those directed towards the root are referred to as **backward** edges. This would give us the graph $T' = (\Gamma, E')$ where E' is the union of the original forward edges and backward edges.

To construct the required adjacency representation of the graph T' we first sort the edges in E' by their first component. The adjacency list can now be constructed by having each edge point to its neighbour in the sorted list on the condition that the two edges have the same first component. The list can be made circular by using the pointer jumping algorithm to find the last node in the adjacency list of a task and

Edge	<1,2>	<2,1>	<1,3>	<3,5>	<5,10>	<10,5>
σ	<2,1>	<1,3>	<3,5>	<5,10>	<10,5>	<5,3>
Edge	<5,3>	<3,6>	<6,3>	<3,1>	<1,4>	<4,7>
σ	<3,6>	<6,3>	<3,1>	<1,4>	<4,7>	<7,8>
Edge	<7,8>	<8,9>	<9,8>	<8,7>	<7,4>	<4,1>
σ	<8,9>	<9,8>	<8,7>	<7,4>	<4,1>	<1,2>

Table 4.1: **Euler tour for the tree T defined in Figure 4.1**

letting the last node point to the first one. To have the pointers between an edge and its reversal we first sort the edges lexicographically on $\{\min(T_i, T_j), max(T_i, T_j)\}$. We can then easily add the pointers between the edge and its reversal.

The above construction takes $O(\log^2 n)$ time using $O(n/\log n)$ processors or $O(\log n)$ time using $O(n)$ processors depending on the sorting algorithm used.

Once we have the graph $T' = (\Gamma, E')$ defined by the adjacency lists of its tasks with the additional pointers as described previously, we can construct the Euler path of T' in $O(1)$ time using $O(n)$ operations on the EREW PRAM model, where $|\Gamma| = n$. We would from now on refer to this Eulerian path for the graph T' as the Eulerian tour defined for the tree T.

The Euler tour of the tree T defined in Fig.4.1 is specified by the successor function shown in Table 4.1. The adjacency representation of the tree is shown in Fig. 4.2.

Having constructed the Eulerian tour of the tree we can now compute the release time of each task $T_i \in \Gamma$. The release time of a task T_i is equal to its depth in the

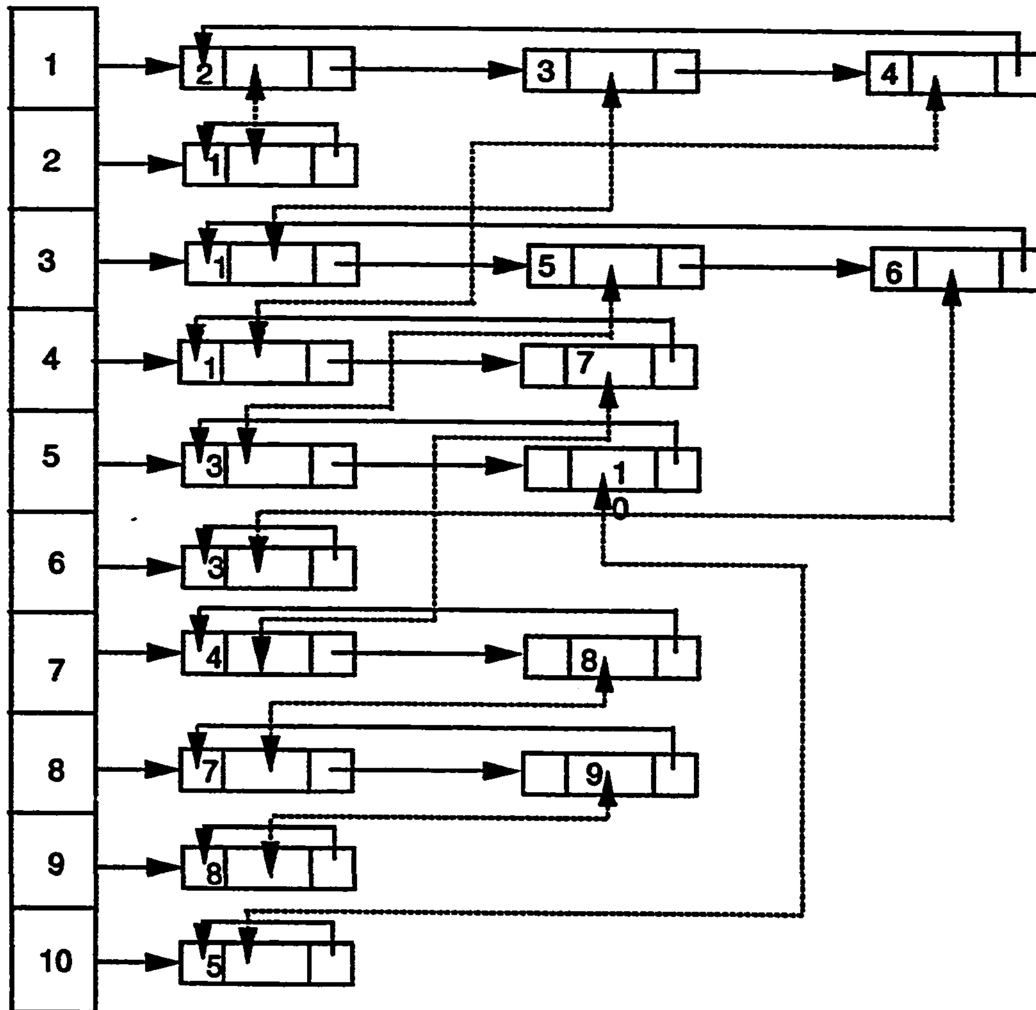


Figure 4.2: The Euler representation of the out-tree

Algorithm Depth_Tree
Input:

i) A tree T defined by the adjacency list L of its tasks.

ii) An Eulerian tour defined by the successor function σ.

iii) A special root task r and for each task $T_i \neq r$ its parent $p(T_i)$.

Output:

i) For each task T_i its depth $\pi(T_i)$.

Procedure Depth_Tree (L,σ,p,π)
begin

i) Set the weights $w(<p(T_i),T_i>) = 1$ and $w(<T_i,p<T_i>) = -1$

ii) Perform parallel prefix on the list defining the Euler path.

iii) The depth of the task T_i $\pi(T_i)$, is equal to the prefix sum of the edge $<p(T_i),T_i>$. The depth of the root $r = 0$.

end.

Figure 4.3: **Parallel algorithm for computing depths of task nodes of a tree**

tree T, i.e. $r(T_i) = \pi(T_i)$. The **depth** $\pi(T_i$ of a task T_i in a tree T=(V,E) is the length of the longest path from the root r to T_i. The procedure for computing the depth of a task T_i is shown in Fig. 4.3.

The computation for the depths of the tasks defined in Figure 4.1 is shown in Table 4.2. The depth calculated for each task T_i is shown in Table 4.3.

The above algorithm runs in $O(\log n)$ time using $O(n)$ processors. The cost can be reduced to $O(n)$ without increasing the time complexity of $O(\log n)$ if the optimal

Edge(e)	$<1,2>$	$<2,1>$	$<1,3>$	$<3,5>$	$<5,10>$	$<10,5>$
Wt(e)	1	-1	1	1	1	-1
$\Sigma wt(e)$	1	0	1	2	3	2
Edge(e)	$<5,3>$	$<3,6>$	$<6,3>$	$<3,1>$	$<1,4>$	$<4,7>$
Wt(e)	-1	1	-1	-1	1	1
$\Sigma wt(e)$	1	2	1	0	1	2
Edge(e)	$<7,8>$	$<8,9>$	$<9,8>$	$<8,7>$	$<7,4>$	$<4,1>$
Wt(e)	1	1	-1	-1	-1	-1
$\Sigma wt(e)$	3	4	3	2	1	0

Table 4.2: **Computing the depths of the tasks**

T_i	1	2	3	4	5	6	7	8	9	10
$\pi(T_i)$	0	1	1	1	2	2	2	3	4	3

Table 4.3: **The Computed depths of the tasks.**

parallel prefix algorithm is used.

Once the depth of the tasks are known the release times of the tasks can easily be determined (see Table 4.4).

In order to compute the deadlines of the tasks we note that we need to compute the heights of the tasks and their ranks in the Eulerian tour defined for the tree T. In the following paragraphs we describe the computations involved in computing the ranks of the tasks in the Eulerian tour of the tree T. An algorithm for computing

Task	1	2	3	4	5	6	7	8	9	10
Release Time	1	2	2	2	3	3	3	4	5	4

Table 4.4: **The Release times of the tasks**

Edge(e)	$<1,2>$	$<2,1>$	$<1,3>$	$<3,5>$	$<5,10>$	$<10,5>$
Wt(e)	1	0	1	1	1	0
$\Sigma wt(e)$	1	1	2	3	4	4
Edge(e)	$<5,3>$	$<3,6>$	$<6,3>$	$<3,1>$	$<1,4>$	$<4,7>$
Wt(e)	0	1	0	0	1	1
$\Sigma wt(e)$	4	5	5	6	7	8
Edge(e)	$<7,8>$	$<8,9>$	$<9,8>$	$<8,7>$	$<7,4>$	$<4,1>$
Wt(e)	1	1	0	0	0	0
$\Sigma wt(e)$	9	9	9	9	9	9

Table 4.5: **Computing the ranks of the tasks**

T_i	1	2	3	4	5	6	7	8	9	10
pre (T_i)	1	2	3	7	4	6	8	9	10	5

Table 4.6: **The Computed ranks of the tasks**

the heights of the tasks follows that:

The order in which the nodes are visited in an Eulerian tour for the tree T is equal to their preorder ranks. The algorithm for ranking the tasks is shown in Fig. 4.4.This algorithm runs in $O(\log n)$ time using $O(n)$ processors. The computation for the ranks of the tasks defined in Figure 4.1 are shown in table 4.5. The ranks computed for each task T_i are shown in Table 4.6.

In computing the heights of the tasks we observe that for every task T_i in the tree T :

$$h(T_i) = \max_{T_j \in \text{succ}(T_i)} \{\pi(T_j)\} - \pi(T_i) \tag{4.1}$$

where $h(T_i)$ is the height of the task T_i, and succ(T_i) is the set containing T_i and all its successors.

Algorithm Ranking the tasks
Input:

i) A tree T defined by the adjacency list L of its tasks.

ii) An Eulerian tour defined by the successor function σ.

iii) A special root task r and for each task $T_i \neq r$ its parent $p(T_i)$.

Output:

i) For each task T_i its preorder traversal rank, pre(T_i).

Procedure Ranking(L,σ , p, pre)
begin

i) Assign to each edge of the form $< p(T_i), T_i >$ and $< T_i, p(T_i) >$ a weight of 1, and 0 respectively.

ii) Perform parallel prefix sum on the list of edges defining the Eulerian path.

iii) Set pre(T_i) to be equal to the prefix sum $< p(T_i), T_i > + 1$. Pre(r) is set to 1.

end.

Figure 4.4: Parallel algorithm for ranking tasks

Define the **Euler Array A** to be the ordered set of vertices obtained by replacing each arc $< T_i, T_j >$ in the Euler tour of T by the task T_j and inserting the root at the very beginning of the array. Define the **Value array** as the depth of each element of A i.e. Value=π(A). Clearly, if $|\Gamma| = n$ then the size of A and Value is 2n-1. The array A and Value can be constructed in constant time if we are given the Euler tour of T and the depth of each task.

Let $l[T_i]$ and $r[T_i]$ be the indices of the leftmost and rightmost appearances of task T_i in A. To compute $l[T_i]$ and $r[T_i]$ for $T_i \neq r$ we do the following:

Given the array A, the element $A[j] = T_i$ is the leftmost appearance of T_i, i.e. $l[T_i] = j$ iff $\pi(A[j-1]) = \pi(A[j]) - 1$. The rightmost appearance of T_i is k i.e., $r[T_i] = k$, iff $A[k] = T_i$ and $\pi(A[k+1]) = \pi(A[k]) - 1$. Note that $l[T_i] \leq r[T_i]$. The leftmost and rightmost appearance of the root is always 1 and 2n-1 respectively, i.e. l[r]=1, and r[r]=2n-1.

Define Range(T_i) $= \{A[l[T_i]], A[l[T_i]] + 1, \ldots A[r[T_i]]\}$ as the list containing all the tasks in the subtree rooted at T_i. Eq. 4.1 can now be restasted as follows:.

$$h(T_i) = \max_{T_j \in Range(T_i)} \{\pi(T_j)\} - \pi(T_i).$$

Thus, our main objective is reduced to finding the maximum of the depths over all ranges Range(T_i) in parallel. We will use the doubling method for our computation.

Assuming that 2n-1 processors are available to us the algorithm for computing thr height of a task is shown in Fig. 4.5.

Algorithm Computing Height of task
Input:

i) The Euler array A and the value array containing the depth of each task in A.

ii) The leftmost occurrence $l[T_i]$ and the rightmost occurrence $r[T_i]$ for each task T_i of A.

iii) An array SPNODE s.t. SPNODE[i]=TRUE if A[i] is the leftmost occurrence of task T_i i.e. $l[T_i] = A[i]$.

Output:

i) The height $h(T_i)$ of each task T_i in the tree T.

```
Procedure Height (A,l,r,Γ,SPNODE,h)
begin
step 1:
        For i:=1 to 2n-1 dopar
                Link(i):=i+1;
        odpar

        Link[2n-2]=Nil;
        For i:=1 to 2n-1 dopar
                Dupvalue[i]:=Value[i];
                Iteration[i]:=0;

        odpar
step 2

        For i:=1 to 2n-1 dopar
                if (SPNODE[i]=TRUE)
                        x=A[i];
                        noelt[x]:=r[x]-l[x]+1;
                        Critic[i]:=⌊log (noelt[x]) ⌋;
                else Critic[i]:=0;
                fi

                delta[i]:=noelt[i]-2^Critic[i]
        odpar
```

Figure 4.5: Parallel algorithm for computing heights of a task in a tree

step 3

```
For i:=1 to 2n-1 dopar
        While (Link[i] ≠ NIL)
              If ((SPNODE[i]=TRUE) and (iteration[i]=Critic[i]))
                    h[x]:=max(Value[i],Dupval[i+delta[i]])-π[i];
              fi

              Iteration[i]:=Iteration[i]+1;
              Value[i]:=Max(Value[i],Value[Link[i]]);
              Dupval[i]:=Value[i];
              Link[i]:=Link[Link[i]];

        end while
  odpar
end.
```

Figure 4.5: Continuation of the parallel algorithm for computing heights of a task in a tree

T_i	1	2	3	4	5	6	7	8	9	10
h(T_i)	4	0	2	3	1	0	2	1	0	0

Table 4.7: **The heights of the tasks defined in Figure 4.1**

The algorithm for computing heights, runs in $O(\log n)$ time using $O(n)$ processors. Its cost can be reduced to $O(n)$ if we convert the tree to a binary tree and then apply the arithmetic evaluation algorithm with the operation **Max** on the resulting binary tree [33]. The heights of the tasks for the tree defined in Fig. 4.1 are shown in Table 4.7. Once we have for a task T_i its height $h(T_i)$ and its rank pre(T_i), we can construct the list L_d sorted according to the tuple $(-h(T_i), \text{pre}(T_i))$ in $O(\log n)$ time using $O(n)$ processors or in $O(\log^2 n)$ time using $O(n/\log n)$ processors. We then evaluate the deadline of a task T_i to be equal to n + the index of T_i in L_d. The deadlines of the tasks defined in Fig. 4.1 are shown in Table. 4.8.

Task	1	2	3	4	5	6	7	8	9	10
Deadline	11	17	13	12	15	19	14	16	20	18

Table 4.8: **Deadlines of the tasks defined in Figure 4.1**

4.3.4 Scheduling Tasks with Release Times and Deadlines

We now have in our hand a set Γ of n-unit length tasks with each task T_i having a non-negative integer release time, $r(T_i)$, and a positive integer deadline $d(T_i)$. Our problem is how to schedule these tasks in parallel so that the resulting ED-schedule is a height priority schedule of the original outforest.

When the number of processors m=1 and all the tasks have the same release time, ED-schedule can be constructed sequentially by scheduling the tasks in non-decreasing order of deadlines.

For the case when $m \geq 1$ and the tasks do not have the same release time, an optimal schedule is obtained by assigning tasks to time slots. At each time interval [i,i+1] a maximum of m tasks can be scheduled. Let A be the set of tasks available when slot [i, i+1] is to be scheduled (the set of available tasks in the interval [i,i+1] consists of all tasks not yet selected that have a release time less than or equal to i) . If $|A| \leq m$ then all the available tasks are processed . If $|A| > m$, then m slots at a time are scheduled. [28], [4].

The algorithm mentioned earlier is readily seen to be highly sequential. Decisions concerning time slot [i,i+1] cannot be made unless the tasks that are available at this time are known. This, of course, depends on which tasks were selected for the

earlier time slots. Thus, a straightforward adaptation of the algorithm would need n steps. The overall complexity of the resulting parallel algorithm is $\Omega(n)$. This is not exactly the complexity we were originally after, since it is almost equal to the complexity of scheduling an outforest sequentially. We are really interested in algorithms with a time complexity of $O(log^k n)$ for some k.

One method of finding a release-deadline schedule for the above task model in parallel is presented in [18]. There the equivalence between finding a schedule in which the release times and deadlines are not violated and finding a perfect matching for a convex bipartite graph is used to find a release time deadline schedule. Their algorithm finds a perfect bipartite matching (thus a release time deadline schedule) in $O(\log n)$ time using $O(n^2)$ processors.

Here we present another method to solve the release-time deadline schedule problem [16].

Our algorithm will consist of three main steps, namely preprocessing, a bottomup pass and, then finally a topdown pass. We will refer to the algorithm as the parallel ED-schedule algorithm.

The first step in the proposed algorithm is to sort the tasks by release times into nondecreasing order. Tasks with the same release time are sorted into nondecreasing order of deadline times. Let $R_1, R_2, \ldots R_k$ be the k distinct release times of the n tasks s.t. $R_1 < R_2 < \ldots < R_k$. Let $R_{k+1} = \infty$.

Next, a complete binary computation tree with k leaves is associated with the

problem. With each node in the tree we associate a time interval (t_l, t_r). The ith leaf node (assuming that the leaf nodes are numbered 1 through k, left to right) is associated with the interval $(R_i, R_{i+1}), 1 \leq i \leq k$. For a nonleaf node N_{internal}, $t_l(N_{\text{internal}}) = t_l(\text{left child of } N_{\text{internal}})$ and $t_r(N_{\text{internal}}) = t_r(\text{ right child of } N_{\text{internal}})$.

Definition 4.3.6 *Let the interval associated with a node N of the tree be* (t_l, t_r). *Define the set of* **available tasks** $A(N)$ *for N to consist of exactly of those tasks that have a release time greater than or equal to* t_l *and less than* t_r. *The set of* **used tasks** *consists exactly of those tasks that will be scheduled between* t_l *and* t_r *for the task set defined by the task set* $A(N)$. *The remaining tasks makeup the* **transferred set.**

In the **bottom-up pass**, a pass is made level by level towards the root. Here the used set and transferred sets for each of the nodes in the computation tree are determined. For a leaf node N_{leaf}, the used set consists of the first $m(t_r - t_l)$ tasks in $A(N_{\text{leaf}})$, i.e. the $m(t_r - t_l)$ tasks with the least deadline times.

For a nonleaf node N_{internal}, the used and transferred sets are computed from the used and transferred sets of its children. Let U_l, U_r, T_l, and T_r be the used and transferred sets for its left and right children respectively. Let the interval associated with node N_{internal}, its left child and right child be (t_l, t_r), (t_l^1, t_r^1), and (t_l^2, t_r^2) respectively. It is clear that $t_l = t_l^1, /mboxt_r = t_r^2$ and $t_r = t_l^2$. The tasks to be scheduled from t_l to t_r^1 are in U_l while some subset of $T_l \cup U_r$ will constitute the set of tasks scheduled from t_r^1 to t_r. Denote Q as the first $m(t_r - t_r^1)$ tasks of $T_l \cup U_r$

that have the least deadline times. It is not difficult to see that Q is the subset of tasks of $A(N_{\text{internal}})$ that is scheduled by sequential ED-schedule algorithm in the interval t_r^1 to t_r. To obtain the set Q we first merge U_r and T_l and then select the first $m(t_r - t_r^1)$ tasks from the merged list. The set Q is then merged with U_l to obtain the used set in nondecreasing order of deadline times. The transferred set for node N_{internal} is obtained by merging the tasks set in $T_l \bigcup U_r$ not belonging to Q with T_r in non-decreasing order of deadline times. The bottom-up pass for the set of tasks defined in Figure 4.1 is shown in Figure 4.6.

In the **top-down pass**, the used sets are updated so that the used set for a node representing the interval (t_l, t_r) is precisely the subset of the tasks that is scheduled in this interval by the ED-Schedule problem for the entire task set. This is done by working down the computation tree level by level starting with the root. We start with the root by marking its used set as being updated. However it is left unchanged. If N is a node whose used set has been updated then the used sets for the left child and the right child of N are obtained as follows:

Let the interval associated with N be (t_l, t_r) and let the interval associated with its left child be (t_l, t_r^1). Let V be the set of tasks in the used set of N with a release time less than t_l. V is obtained by first selecting the tasks in the used set of N with release times less than t_l and then compacting the selected elements without changing the relative order . This could be performed in $O(\log|V|)$ time [43]. The new used set U' for the left child of N is computed by first merging V and U in a set W and then selecting the first $m(t_r^1 - t_l)$ tasks with the least deadlines from the set W. The used set for the right child of N consists of all tasks in the used set of

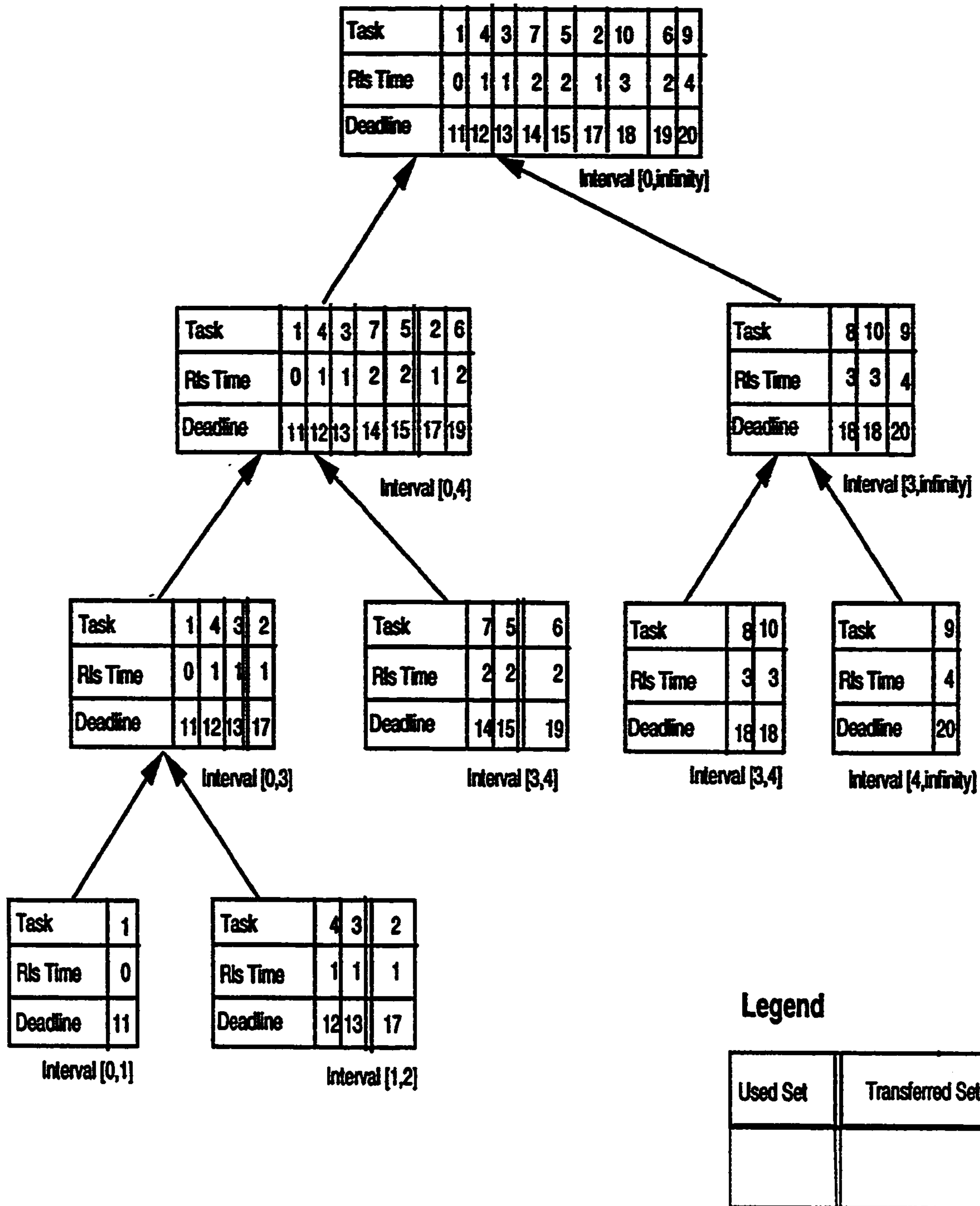


Figure 4.6: Computing release deadline-times : The bottom up pass

N that are not included in U'.

The top-down pass for the set of tasks defined in Figure 4.1 is shown in Figure 4.7.

The proof that the above procedure can correctly produce a ED-schedule for a set of tasks specified by their release times and deadlines is shown in [16]. To compute the time and processor complexities of the above procedure we note that the first step namely the preprocessing step can be performed in $O(\log n)$ time using $O(n)$ processors or $O(log^2 n)$ time using $O(n/\log n)$ processors depending on the sorting algorithm used.

Each of the bottom-up pass and top-down pass requires $O(\log n)$ iterations . In each of the pass over the computation tree we are essentially performing a fixed number of merges of ordered sets at each node.

Merging of two sets containing $p+q$ elements respectively as mentioned in sec. 3.5 can be performed in $O(\log(p+q))$ time using $O(\frac{p+q}{\log(p+q)})$ processors. Thus the **ED- schedule** for a set of tasks Γ, defined by release times and deadlines can be constructed in $O(\log^2 n)$ time using $O(n/\log n)$ processors.

From section 4.3.2 and section 4.3.3 we find that a height priority schedule for an out-tree $T = (\Gamma, E)$ where each of the tasks $T_i \in \Gamma$ has a unit execution time, can be computed in $O(log^2 n)$ time using $O(n)$ processors. We can reduce the cost complexity to $O(n \log n)$ without increasing the time complexity if we use the optimal algorithm for parallel prefix, the arithmetic evaluation algorithm for computing

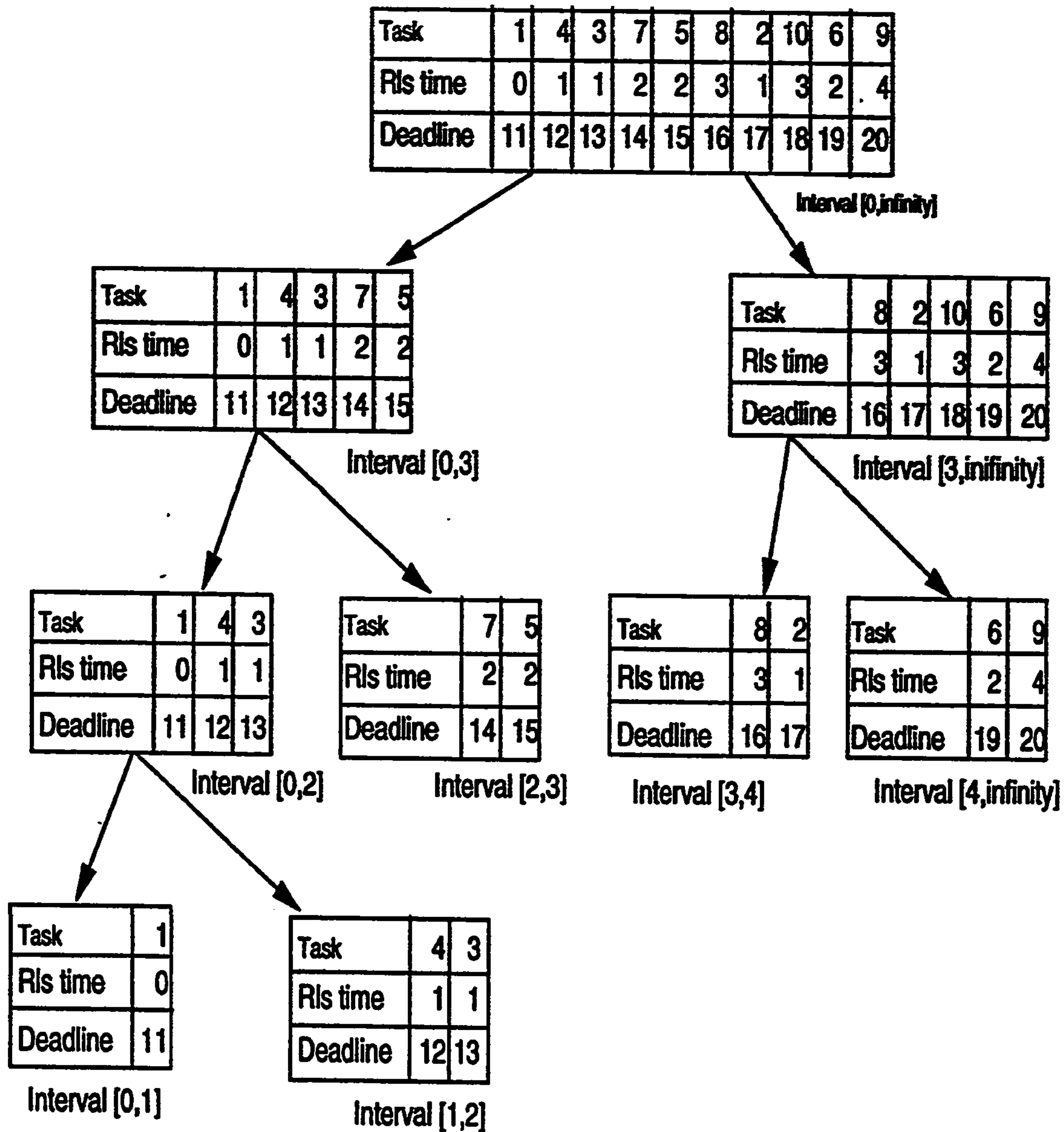


Figure 4.7: Computing release deadline-times : The top down pass

the heights, the $O(log^2 n)$ merge-sort algorithm for sorting.

If we use the pipelined-merge sort algorithm for sorting and Dolev's $O(\log n)$ time algorithm time to find a release time deadline schedule then we can obtain a height priority schedule for T in $O(\log n)$ time using $O(n^2)$ processors.

4.4 Scheduling UET task graphs in parallel

The task model assumed for the scheduling problem in the last section was restricted to a tree. In this section we will study the parallel scheduling of an application in which this restriction is removed. Here, the application is represented by an arbitrary precedence graph $G = (\Gamma, E)$ where $\Gamma = \{T_1, T_2, \dots T_n\}$ is a set of unit-length tasks and E is the set of edges representing the precedence constraints between the tasks on m homogeneous application processors. We will refer to such a directed acyclic graph (DAG) representing the application as a **UET task graph**.

This problem has been shown to be NP-complete for an arbitrary number of m processors [14]. For a fixed $m \geq 3$ no algorithms which ensure optimal schedules are yet known. For $m = 2$ the problem is solvable using the Coffman-Graham algorithm [13]. A greedy sequential heuristic to the problem has been proposed by Graham [25]. The schedule length produced by the heuristic is shown to be $(2 - 1/m)$ from the optimal. An outline of the heuristic is shown in Fig. 4.8.

In this section our objective is to define a nongreedy depth schedule which ap-

Algorithm SEQ_UET_SCH

Input:

i) A set Γ consisting of n tasks with unit execution times.

ii) The number of processors m.

Output:

i) For each task $T_i \in \Gamma$, its starting time $s(T_i)$, and the processor $p(T_i)$ to which it has been assigned.

```
Procedure SEQ_UET_SCH(Γ, n, m, ps)
begin
    U:=Γ      /* set of unassigned tasks */
    c:=0; /*empty time slot */
    While U ≠ φ do

        R:={t ∈ Γ | ∄t' ∈ U such that t' precedes t };
        /* R is the ready set i.e. the set of tasks which are
        available for processing */
        i:=1; /* Number of processors assigned */
        while R ≠ φ and i ≤ m do

            Choose a task t from R;
            Allocate to processor P_i for time c the task t;
            R:=R-t
            U:=U-t
            i:=i+1;

        end while;

        c:=c+1;
    end while;
end.
```

Figure 4.8: Algorithm for Sequential UET scheduling

proximates the optimal schedule by the same factor [18].

Definition 4.4.1 *Define the depth $\pi(T_i)$ of a task T_i in G to be equal to the length of the longest path from a starting task node in G to T_i.*

Given the depths of the tasks in Γ we can define the **depth schedule** for a dag G by the following procedure.

for k:=0 to $\pi(G)$ do

– schedule the d_k tasks of G of depth k in the next $\lceil \frac{d_k}{m} \rceil$ time slots;

where $\pi(G)$ is the depth of the DAG G and d_k is the number of tasks in G of depth k.

Let the length of the m processor depth and optimal schedule be $D_m(G)$ and $O_m(G)$ respectively.

Theorem 4.4.1 $\frac{D_m(G)}{O_m(G)} \leq 2 - 1/m$

Proof

Assume that there is a precedence graph for which the theorem is false. Let i be the number of idle periods in an optimal schedule for G. Construct a new graph G' by adding i independent tasks to G.

Clearly, $D_m(G') \geq D_m(G)$ and $O_m(G) = O_m(G') = n/m \geq \pi(G')$, where n is the number of tasks in G'. Since $D_m(G')/O_m(G') \geq D_m(G)/O_m(G)$ it follows that G' also contradicts the theorem. Let f and e be the number of slots in a depth schedule for G' containing m tasks and at least one but less than m tasks respectively.

We observe that $e \leq \pi(G')$ and $f \leq (n - \pi(G'))/m$. The following inequalities lead to a contradiction:.

$$
\begin{aligned}
\frac{D_m(G')}{O_m(G')} &= \frac{e+f}{n/m} \\
&\leq \frac{\pi(G') + (n - \pi(G'))/m}{n/m} \\
&= \frac{\pi(G')(m-1+n)}{n} \\
&\leq \frac{(n/m)(m-1)+n}{n} \\
&= 2 - \frac{1}{m} \square
\end{aligned}
$$

4.4.1 Computing the depth of a DAG

Given an acyclic directed graph $G = (\Gamma, E)$ where $\Gamma = \{T_1, T_2, \dots T_n\}$ represents the set of tasks and E represents the precedence relationships between the tasks, we are required to compute the depth of each task $T_i \in \Gamma$. To compute this we need to know the length of the longest path from a starting task node in G to task T_i. In the following paragraphs we describe the steps involved in computing the **All Pairs Longest path** in a graph i.e. the longest paths between all pairs of nodes.

Definition 4.4.2 *Let each element A_{ij} of the* **adjacency matrix** *A of graph G be defined as follows:*

$$A_{ij} = 1 \text{ if } <i,j> \in E,\ A_{ij} = -\infty \text{ if } <i,j> \notin E \text{ and } A_{ij} = 0 \text{ if } i{=}j.$$

Also let A^k_{ij} denote length of the longest path from task T_i to T_j going through at most k intermediate tasks. Then the **closure matrix** *C of graph G can be defined as an $n \times n$ matrix, such that C_{ij} is the length of the longest path from task T_i to T_j. It is clear that $C_{ij} = A^n_{ij}$.*

Given the matrix $A^{k/2}$ we can compute A^k as follows:

$$A^k_{ij} = \max_l \{A^{k/2}_{il} + A^{k/2}_{lj}\}.$$

A parallel algorithm which can compute A^k from $A^{k/2}$ is shown in Figure 4.9. Since the structure of the algorithm is similar to a matrix multiplication algorithm we will refer it to as **PRAM_MATRIX_MULT**.

This algorithm can be executed on the EREW PRAM model in $O(\log n)$ time using $O(n^3)$ processors. If we apply Brent's scheduling theorem to compute the maximum then we can achieve the same time complexity using $O(n^3/\log n)$ processors.

ALGORTIHM PRAM_MATRIX_MULT

Input

i) Two $n \times n$ matrices A and B.

Output

i) The matrix C s.t.

$$C_{ij} = \max_{l}\{A_{il} + B_{lj}\}.$$

Procedure PRAM_MATRIX_MULT(A,B,C)
begin
Step 1

Use the **broadcast** algorithm to make n copies of matrices A and B. Processor $P(i,j,k)$ contains the kth copy of A_{ij} and B_{ij}.

Step 2

```
for j:=0 to n dopar
        for i:=1 to n dopar
                for k:=1 to n dopar
                        C2_ijk := (A_ikj + B_kji);
                odpar
        odpar
odpar
```

(In the printed code: $C2_{ijk} := (A_{ikj} + B_{kji});$)

Step 3

Compute the maximum $C_{ij} := \max_k\{C2_{ijk}\}$

end.

Figure 4.9: Parallel PRAM_MATRIX_MULT algorithm

Algorithm CONS_DEPTH_DAG

Input:

i) The adjacency matrix A for the graph G.

ii) The set of tasks $\Gamma = \{T_1, T_2, \ldots T_n\}$

Output:

i) The depth $\pi(T_i)$ for each task $T_i \in \Gamma$.

Procedure CONS_DEPTH_DAG(A,π)
begin
Step 1:

```
For k:=1 to ⌈log(n)⌉ do
        PRAM_MATRIX_MULT(A,A,C);
end for
```

Step 2:

Using the procedure **MAX** compute

$$\pi(T_i) := \max_{1 \le j \le n} \{A_{ij}\}$$

end.

Figure 4.10: **Parallel algorithm for computing depths of nodes in a DAG**

Now if we have the closure matrix C then we can compute the depth of each task $T_i \in \Gamma$ using the following relation:

$$\pi(T_i) := \max_{1 \le j \le n} \{C_{ij}\}$$

The algorithm for computing the depth of a task $T_i \in \Gamma$ is shown in Fig. 4.10. Note that *Step 1* of the algorithm implements the **ALL PAIRS LONGEST PATH ALGORITHM.**

The time complexity of the **CONS_DEPTH_DAG** algorithm is based upon the time complexity of *Steps 1 and 2*. We note that the number of iterations in *Step 1* is $O(\log n)$ and that each iteration calls the procedure **PRAM_MATRIX_MULT (A,A,C)** which has a complexity of $O(\log n)$. Thus, total time complexity of *step 1* is $O(\log^2 n)$. The time complexity of *Step 2* of the algorithm is clearly $O(\log n)$. With this, the overall time complexity of **CONS_DEPTH_DAG** is $O(log^2 n)$ using $O(n^3)$ processors. Using **Brent's scheduling theorem** to compute the maximum, the cost can be reduced to $O(n^3 \log n)$.

4.4.2 The parallel depth scheduling algorithm

Once we know the depth of the task graph G and the depth of each task, $T_i \in \Gamma$ we can easily construct the depth schedule using the steps shown in Figure 4.11. Notice that in this procedure we are assuming that the application processors are numbered $1 \ldots n$ and that the time slots start from time unit 0.

The time complexity of each of the steps 2,3,4 in the above procedure is $O(\log n)$ using $O(n/\log n)$ processors. Step 1 takes $O(\log n)$ time using $O(n)$ processors or $O(\log^2 n)$ using $O(n/\log n)$ processors. See section 3.5 . The overall complexity of constructing a depth schedule from a set of tasks Γ and an adjacency matrix A remains $O(log^2 n)$ using $O(n^3/\log n)$ processors.

ALGORITHM Depth_Schedule
Input

i) A set of tasks $\Gamma = \{T_1, T_2, \ldots T_n\}$.

ii) The depth $\pi(T_i)$ of each task $T_i \in \Gamma$.

iii) A set of m application processors.

Output

i) For each task $T_i \in \Gamma$ the starting time $s(T_i)$ and the processor $p(T_i)$ to which T_i has been assigned.

Procedure Depth_Schedule(Γ,s,p)
begin

i) Sort the tasks according to their depths
into an array TASKAR.

ii) Using the parallel segmented prefix computation algorithm
find for each task TASKAR[j], the number of tasks that have
lower indices in TASKAR than TASKAR[j,] and have
the same depth as TASKAR[j].
Assign the value resulting from the computation to REM[j].

Figure 4.11: Parallel Depth scheduling algortihm

---

iii) Compute in parallel the number of slots, SLOTS[k]
required to schedule the d_k tasks of depth k.
This can be computed by first finding the task TASKAR[i]
of depth k which has the maximum REM[i]. Then, we assign
SLOTS[k] = $\lceil$REM[i]$/m\rceil$.

iv) Use the parallel prefix sum algorithm
to find for each depth k the number of slots
$BS[k]$ required to schedule the tasks of depth less than k.

v) Schedule a task TASKAR[i] of depth k at the time slot
which starts at BS[k] + $\lfloor$REM[i]$/m\rfloor$.
It is assigned to the processor whose id is equal to
REM[i] mod $m + 1$.

end.

Figure 4.11: **Continuation of parallel Depth scheduling algortihm**

Chapter 5

Par_ETF Algorithm and its Design

5.1 Problem Formulation

From the last chapter, we note that most of the existing parallel scheduling algorithms have been designed to parallelize the *classical scheduling algorithms*. Many of these algorithms assume that the application model and the underlying computing system do not have any inter-task and inter-processor communication overheads. These assumptions however are not valid for the existing message-passing multiprocessor systems or computer networks, where inter-task and interprocessor communication overheads are an important aspect and are, therefore, not negligible.

In this and the following chapters we present our solution to the above problem.

We have, as part of our thesis work, looked into the existing sequential scheduling algorithms where the interprocessor communication overheads have been taken into account, and have selected the ETF(**Earliest Task First**) heuristic for parallelization [31].

Our reasons for choosing ETF for parallelization include:

- The underlying task and system model assumed by ETF can be used to represent several types of computer systems.
- It has a large sequential time complexity which can be reduced by a considerable amount through parallelization.
- ETF is based on an extensive theoretical study and its schedules, through worst case analysis have been shown to be bounded by the sum of Graham's bound for list scheduling and the communication requirement over some immediate predecessor-immediate successor pairs along one chain.

5.2 ETF heuristic: A general outlook

In the ETF heuristic the intertask and interprocessor communication overhead is made part of the problem formulation.

The application is assumed to be given as a set $\Gamma = \{T_0, T_1, T_2, T_3 \cdots T_{n-1}\}$ of tasks, where the execution time of each task $T_i \in \Gamma$ is represented by a_i.The precedence constraints between the tasks are represented by the set of edges E.

Associated with each edge (T_i, T_j) is a positive integer $c(T_i, T_j)$ which denotes the number of messages sent from T_i to T_j upon the termination of T_i. We refer to T_i as the **immediate predecessor** of T_j and T_j as the **immediate successor** of T_i. This task model consisting of task nodes and edges in the task forms an "**enhanced**" **directed acyclic graph (DAG)** and is denoted by a quadruple $G = (\Gamma, E, a, c)$.

The underlying application processor system is represented by a tuple $S = S(m, l)$ where m is the number of identical processors and $l(P_u, P_v)$ is a parameter to represent the time needed to transfer a message unit from processor P_u to P_v. We assume that the communication subsystem is contention free, i.e., the time to take $c(T_i, T_j)$ unit of messages from P_u to P_v is : $l(P_u, P_v) \times c(T_i, T_j)$.

The ETF heuristic is an extension to an earlier heuristic proposed by Rayward-Smith in which the computing model was confined to unit communication (UCT) and unit execution time (UET) [53]. The *generalized list scheduling* method used there adopts the same greedy strategy as Graham's list scheduling. A task T_i can be processed on the processor P_u at time t if T_i has no immediate predecessors, or each immediate predecessor of T_i has been scheduled to start on processor P_u at time $\leq t - 1$ or on processor P_v,$P_u \neq P_v$, at time $\leq t - 2$.

ETF uses a similar greedy strategy: the earliest schedulable task is scheduled first. The heuristic is event-driven. At each step, the algorithm schedules a task $\hat{T}$ on a processor $\hat{P}$, if the earliest starting time of $\hat{T}$ on $\hat{P}$ is the smallest among all the **schedulable tasks** and all the **available processors**.

5.3 Design of the Sequential ETF heuristic

The input to the ETF heuristic is the task and system model while in its output we have for each task $T_i \in \Gamma$ the **starting time** $s(T_i)$, **finishing time**, $f(T_i)$ and the **processor** $p(T_i)$ to which T_i has been assigned.

The starting time of a task T_i is determined by several factors namely:

- The time its preceding tasks finish. Let the set of preceding tasks of T_i be denoted by $D(T_i)$.
- The length of the communication delays.
- The processors to which T_i and its predecessors are allocated.

Let $r(T_i, P_u)$ denote the time the last message for task T_i arrives at the possible hosting processor P_u. It can be calculated as follows:

$$r(T_i, P_u) = \begin{cases} 0 \text{ if } T_i \text{ has no predecessors} \\ \max_{T_j \in D(T_i)}\{f(T_j) + c(T_j, T_i) \times l\,(p(T_j), P_u)\} \quad \text{otherwise} \end{cases} \tag{5.1}$$

We will refer to $r(T_i, P_u)$ as the **ready time** of task T_i on processor P_u.

Refer to a task T_i as **available** if all its predecessors have been scheduled. Then the **earliest starting time** $e_s(T_i)$ of an available task T_i is calculated by:

$$e_s(T_i) = max\{CM, \min_{P \in I}\{r(T_i, P)\}\}$$

where CM is the "**current moment**" denoting the current time of the event clock and I is the set of **free processors** at time=CM . The objective of ETF is to find a task $\hat{T}$ from the set A of available tasks and a processor $\hat{P}$ from the set of free processors such that $e_s(\hat{T}) = \min_{T \in A}\{e_s(T)\}$. If there are more than one tasks which have the same minimum ready time then the task with the least id is selected as $\hat{T}$. Similarly, if a selected task $\hat{T}$ has the same minimum ready time on more than one processor then the processor with least processor id is selected as $\hat{P}$.

Let the earliest starting time among all available tasks be $\hat{e_s}$. Then it is clear that:

$$\hat{e_s} = e_s(\hat{T}) = max\{CM, r(\hat{T}, \hat{P}\}\}$$

We note that since there are arbitrary communication delays in the model a newly available task after the event clock has been advanced may have an earlier starting time than that of the selected task $\hat{T}$. To overcome this difficulty a second variable called "**next moment**" and abbreviated as *NM* is introduced to keep track of the scheduling process. It denotes the earliest time after CM at which one or more currently busy processors become free. *NM* is set to ∞ if all processors are free after *CM*. If $NM \geq \hat{e_s}$ then it is clear that the scheduler will not generate any available task that can be started earlier than $\hat{e_s}$. Thus a scheduling decision for this case can safely be made. If it is otherwise, i.e. $NM < \hat{e_s}$ then the event clock is advanced and the decision is postponed. Fig.5.1 shows the sequential ETF algorithm as proposed by Hwang et al. [31] .

Step 1 of the **SEQ_ETF** algorithm is basically an initialization step. During this step we construct two sets **ProcAv** and **TaskAv** containing a list of available

Algorithm SEQ_ETF

Input

i) The total number of application tasks n.

ii) An array a representing the execution times of the tasks. The execution time of the task T_i is denoted by $a[i]$.

iii) The Adjacency matrix A representing the precedence relationships between the tasks. An element $A[i,j]$ of A is 1 if task i is a predecessor of task j and is 0 otherwise.

iv) A communication matrix c representing the communication between the tasks. The value of the element $c[i,j]$ represents the number of messages sent from an immediate predecessor i to immediate successor j.

v) The total number of application processors, m.

vi) A matrix l representing the time to transfer messages between processors. Thus $l[u,v]$ represents the time to transfer a message from processor u to processor v.

Output:

For each task $T_i \in \Gamma$

i) its starting time T_i, $s(T_i)$.

ii) its finishing time T_i, $f(T_i)$.

iii) the processor to which it has been assigned $p(T_i)$.

Figure 5.1: **The sequential ETF heuristic: Input and output**

Procedure SEQ_ETF(m,n,a,A,c,l,s,f,p)
begin
Step 1: (Initialize)

Let ProcAv be the set of available processors;
Include all the processors in the application system in ProcAv;
Compute the number of predecessors, NPred[j], of Task T_j;
Let TaskAv be the set of tasks having no predecessors;

Q := 0;
CM:= 0;
NM:= ∞;
For each ((T $\in$ TaskAv) and (P $\in$ ProcAv)) do
r(T,P) := 0;
end for

Step 2:

While $Q < m$

while ((ProcAv $\neq \phi$) and (TaskAv $\neq \phi$) do

Step 2.1.1:

Find $\hat{T} \in$ TaskAv and $\hat{P} \in$ ProcAv such that
temp = $r(\hat{T}, \hat{P}) = \min_{T \in \text{TaskAv}} \min_{P \in \text{ProcAv}} r(T, P)$;
Let $\hat{e}_s$ = max {CM, temp };

Figure 5.1: The sequential heuristic: Steps 1, and 2.1.1

Step 2.1.2:

If $\hat{e}_s \leq$ NM

Assign $\hat{T}$ to run on $\hat{p}$;
$p(\hat{T}) := \hat{p}$;
$s(\hat{T}) := \hat{e}_s$;
$f(\hat{T}) := s(\hat{T}) + a(\hat{T})$;
Remove $\hat{T}$ from TaskAv;
Remove $\hat{P}$ from ProcAv;
Q := Q+1;
if ($f(\hat{T}) \leq$ NM)
NM := $f(\hat{T})$;
fi

else
exit the inner loop;
fi
end while

Step 2.2:
CM := NM;
Compute new NM;

Step 2.3:
For each T, P such that T has finished on P at time moment CM do

Include P in the set of free processors ProcAv;
For each T' s.t. $A[T,T'] = 1$ do
NPred[T']:= NPred[T']-1;
If (NPred[T']=0)
T' is a newly available task;
fi
endfor

endfor

Figure 5.1: The sequential heuristic: Steps 2.1.2, Step 2.2, Step 2.3

Step 2.4:
    For each newly available task T' do
        Include T' in TaskAv;
        For each processor p do

$$r[T',p] := \max_{T \text{ precedes } T'} \{f(T) + c(T,T') \times l(p(T),p)\};$$

        endfor
    endfor
endwhile
end.

Figure 5.1: The sequential ETF heuristic: Step 2.4

processors and available tasks respectively. Each available task T's ready time on an available processor P is initialized to 0, i.e. $r(T,P) = 0$, $\forall T \in$ **TaskAv** and $P \in$ **ProcAv**. A number of auxiliary variables including CM denoting the current value of the event clock, Q denoting the number of scheduled tasks and NM denoting the next moment are also initialized in this step. As for the time complexity of this step, then it can clearly be seen that this step can be executed in $O(n^2 + mn)$ time.

In **step 2** of the algorithm, decisions regarding the scheduling of the tasks are made. The loop in this step is repeated at most n times. This step consists of 4 main substeps. In **step 2.1** the ready times of all the available tasks in all the free processors are compared and the task $\hat{T}$ and processor $(\hat{P})$ are selected such that among all feasible assignments of tasks to processors, $(\hat{T})$ can start on $(\hat{P})$ the earliest. This comparison and selection will take $O(nm)$ time to execute.

Let the possible starting time of $\hat{T}$ on $(\hat{P})$ be $\hat{e}_s$. The decision of whether $\hat{T}$ should be scheduled on $\hat{P}$ depends on the value of $\hat{e}_s$. If $\hat{e}_s$ >NM, then there might

be a newly available task after the event clock is advanced, whose starting time is less than that of the selected task $\hat{T}$. Thus in this case we exit **step 2.1**, advance the event clock, and find the newly available tasks. If it is otherwise i.e. $\hat{e}_s \leq$ NM then the scheduling decision is made. The selected task $\hat{T}$ is removed from the set of available tasks and the selected processor $\hat{P}$ is removed from the set of available processors.

If the selected task $\hat{T}$ would complete before NM then NM is updated to the finishing time of the task. The **substep 2.1** is repeated if there remain available tasks and available processors. We note that for each iteration of **step 2.1**, the algorithm either schedules a task or completes at least one task. Consequently this substep is executed at most $2n$ times. Previously we had noted that each iteration of step 2.1 take $O(mn)$ time. Thus the complexity of step 2.1 is $O(mn^2)$ for at most $2n$ iterations.

Steps 2.2, 2.3, and 2.4 of the algorithm are executed if we can no longer make scheduling decisions in **step 2.1** and there remain some unscheduled tasks in Γ. We start by first advancing the value of event CM to NM and then finding a new value of NM. The new value of NM is equal to the minimum of the finishing times of the tasks which remain uncompleted till the current moment.Finding the new moment takes $O(m)$ time.This is because at any moment at most $O(m)$ tasks are scheduled.

In **step 2.3** we add each processor P that has just finished a task T to the set of available processors. For each task T' whose predecessor task T has just finished, we decrement **NPred**[T'], the number of predecessors of T' that remain uncompleted

by 1. If the number of such predecessors of T' is equal to zero then T' is regarded as a newly available task and is added to the set of ready tasks. Since each task T can have at most $O(n)$ successors and there are at most $O(m)$ tasks that complete at **step 2.3** ,we find that each iteration of **step 2.3** takes $O(mn)$ time to execute.

In **step 2.4** we find for each newly available task T, its ready times on each one of the m application processors. The ready time of a newly available task T on a processor P is calculated using equation 5.1. To compute the time complexity of **step 2.4** we note that finding the ready time of a newly available task T on a processor P takes $O(n)$ time. Thus, it would take $O(mn)$ time to find the ready times of the task T on each one of the m processors. Since there can be at most $O(n)$ newly available tasks, we find that the running time of **step 2.4** in the worst case is $O(mn^2)$. As for the overall complexity of the whole algorithm, we note from our above discussion that is $O(mn^2)$ where m and n are the number of tasks and processors respectively.

As for the schedule produced by the ETF schedule, it was proved that for any EDAG task model $G = (\Gamma, E, a, c)$ to be scheduled on a system $S = (m, l)$, the schedule length ω_{ETF} obtained by ETF always satisfies

$$\omega_{\text{ETF}} \leq (2 - \frac{1}{n}) \times \omega^i_{opt} + C_{max} \tag{5.2}$$

where ω^i_{opt} is the optimal schedule length obtained by ignoring the inter-processor communication and C_{max} is the maximum communication requirement along all chains in Γ [31]. That is,

$$C_{max} = \max\{l_{max} \times \Sigma_{i=1}^{t-1} c(T_{c_i}, T_{c_{i+1}}) : (T_{c_1}, \cdots, T_{c_t}) \text{is a chain in } \Gamma\} \tag{5.3}$$

I	CM(I)	NM(I)	T'	P'	s(T')	f(T')	Decision
1	0	∞	1	1	0	5	made
2	0	5	4	2	0	7	made
3	0	5	None	None			
4	5	7	3	1	5	7	made
5	5	7	None	None			
6	7	∞	5	2	7	12	made
7	7	12	7	1	7	15	made
8	7	12	2	3	15	17	postponed
9	12	15	2	3	15	17	made
10	12	15	None	None			
11	15	17	None	None			
12	17	∞	6	3	17	22	made
13	17	22	None	None			
14	22	∞	8	3	22	27	made
15	22	27	9	2	36	39	postponed
16	27	∞	9	3	27	30	made

Table 5.1: **The scheduling decisions made for the application in fig. 2.1**

The above bound basically, indicates that the length of an ETF schedule is bounded by the sum of Graham's bound for list scheduling and the maximum communication requirement along all chains in Γ. In fact, it has been shown that the bound can further be reduced to the sum of Graham's bound for list scheduling and the communication requirement C along one chain of tasks. The algorithm for calculating this communication requirement C is given in [31].

Table 5.1 illustrates the application of the ETF algorithm on the task and system model given in Fig. 2.1.

5.4 Objectives of Par_ETF

Our primary goal is to design a fast and efficient parallel ETF scheduling algorithm. From this goal we basically want to maximise the parallelism without increasing the cost complexity drastically.

Our second objective is to realize the **Par_ETF** algorithm on a realistic Network model. We had mentioned before that the **PRAM model** is not a realistic model. The network model we have chosen for our algorithm is the **weak hypercube model**. This is a hypercube model in which each processor is allowed to send or receive at most one packet and perform a constant number of local computations in a single time step. Our algorithm would belong to the **normal class** of algorithms in which the communication links are used in only one dimension at a time and consecutive dimensions are used at consecutive time steps. A particular advantage of this class of algorithms is that it can be simulated on a variety of models such as the shuffle-exchange, the cube-connected cycles, or the butterfly with only a constant slowdown [48].

5.5 Basic Operations on the Hypercube Model

Before we go on to describe the design of Par_ETF algorithm let us study how the basic operations are performed on the weak hypercube model. The operations we would be discussing include **Broadcasting, Data Sum, and Prefix Sum.** These operations, as we would note, are the building blocks of our Parallel ETF algorithm.

```
Procedure Broadcast (A,j,k)
begin
        If (j ≠ k)
                A := null;
        fi
        for i := 0 to (d − 1) do
                send A to processor j^(i);
                Receive B from processor j^(i);
                If (A ≠ null)
                        A := B;
                fi
        endfor
end.
```

Figure 5.2: **Broadcasting in hypercubes**

For these algorithms we would use the notation $i^{(b)}$ to represent the number that differs from i in exactly the bit b. Also i_b would be used to refer to the ith bit of the number i.

5.5.1 Data Broadcasting

In data broadcasting we wish to broadcast the data originating in the register A of processor k of a hypercube of dimension d to the remaining PE's of the hypercube. The value of register A in the remaining processors are assumed to be null.

The procedure, shown in Fig. 5.2 is executed by each one of $n = 2^d$ processors in the hypercube. Its time complexity is $O(d) = O(\log n)$.

Lemma 5.5.1 *The data originating in processor k of the hypercube of size n could*

be broadcasted to the remaining processors in the hypercube in $O(\log n)$ *time.*

Proof of Lemma 5.5.1 *Let* $n = 2^d$. *The above lemma can be restated that after* $O(d)$ *iterations the data* A *originating in processor* k *will be broadcasted to all the processors which differ from* k *by at most the first* d *bits. This is because in a hypercube of dimension* d *a processor* k *will differ from any other processor by at most* d *bits. Thus a proof to the above statement will prove the Lemma.*

Let there be a hypercube of dimension d *s.t.* $2^d = n$. *We will use mathematical induction on the number of iterations for the proof. For the basis step it can clearly be seen that after the first iteration, the data* A *originating in processor* k *will be broadcasted by the* broadcast algorithm *to all the processors which differ from* k *in the zeroth bit.*

For the induction step we assume that after $O(s)$ *iterations, the data* A *originating from processor* k *of the hypercube, have been distributed to all of the processors in the hypercube which differ from* k *in the first* s *bits or less. We have to show that after* $O(s+1)$ *iterations the data* A *from processor* k *would be distributed by the* **broadcast procedure** *to all the processors which differ from* k *in the first* $s+1$ *bits or less.*

At the $s+1st$ *iteration each processor* i *in the hypercube will send and receive data from the processor* i^s *which differs from* i *in* s*th bit. If a processor receives a non-null value from its neighbor it would update its register* A. *Otherwise it would do nothing. Since at the end of the* s*th iteration* 2^s *processors had the correct value of* A *and the rest contained null values, we would have in the* $s+1$ *st iteration,* 2^s *new*

```
Procedure Window Broadcast (A,k)
begin
        for i:= to k-1 do
                Send A to processor j^(i)
                Receive B from processor j^(i)
                If (B ≠null)
                        A:=B;
                fi
        endfor
end
```

Figure 5.3: **Window broadcasting in hypercubes**

processors updating their A registers. As a consequence after the $s+1$ *st iteration we would have* $2^s + 2^s = 2^{s+1}$ *processors containing the data originating from processor* k*. It follows from here, that after* $O(d)$ *iterations, all of the* $n = 2^d$ *processors in the hypercube would contain the data A originating from processor* k. □

The above algorithm can be generalized to **window broadcast**. In window broadcast the hypercube is assumed to be partitioned into windows of size 2^k processors each s.t. the processors in each window form a hypercube of dimension k. For the sake of simplicity, assume that the processor indices in each window differ only in their least significant k bits. Our objective is to broadcast a data in A register of a single processor in each window to all other processors in the window. Except for the originating processors all the other processors in the window are assumed to contain null values in their A registers. The algorithm for **window broadcasting** is shown in Fig. 5.3. The time complexity of the **Window broadcast algorithm** is $O(k)$ where k is the dimension of the subhypercube making up the window.

```
Procedure Datasum (A, t, total)
begin
    total := A;
    for i:= 0 to k-1 do
        Send total to processor j^(i);
        Receive B from processor j^(i);
        total := total+B;

    endfor
end.
```

Figure 5.4: **Computing a sum in a hypercube**

5.5.2 Data Sum

Assume that a d dimensional hypercube is partitioned into subhypercubes of dimension k. The **data sum** operation sums the A register data of all the processors in the same window and the result is left in every processor of the window.

The procedure for the **data sum** operation is similar to the **window broadcast procedure**(See Fig. 5.4). It can be performed in $O(k)$ time by first summing in subwindows of size 2, then of size 4, and so on until the subwindow size becomes $w = 2^k$. It is executed by each of the 2^k processors making up the hypercube.

Lemma 5.5.2 *Let A be a register in a hypercube. Using the* **data sum procedure** *all the processors in a window of size $W = 2^k$ can compute the sum of the A registers in $O(k)$ time*

Proof of Lemma 5.5.2 *The proof is similar to the proof given in Lemma 5.5.1* □.

The **Data Sum procedure** described above can be adapted to perform any other associative operation in a hypercube, such as finding a maximum or minimum of a list of values belonging to an ordered set.

5.5.3 Prefix Sum

The assumptions we make in this section are similar to the ones we made in section 5.5.2. Let a processor l, be the qth processor in window i, if $l = iW + q$, $0 \leq q < W$ where W is the size of the window.

Our objective is to compute in processor l the prefix sum S of l using the following equality:

$$S = \Sigma_{j=0}^{q} A(iW + j), \quad 0 \leq i < P/W, \quad O \leq q < W$$

Note that $A(s)$ in the above equality refers to the value of the register A in the sth processor of the hypercube.

The prefix sums in windows of size $W = 2^t$ may be easily computed if we know the following values in each of the size 2^{t-1} subwindows that make up the 2^t window:

i) Prefix sums in the 2^{t-1} subwindow,

ii) Sum of all A values in the 2^{t-1} subwindow.

```
Procedure Prefix Sum (A,k,S)
begin
        S := A(j);
        T := A(j);
        for b:=0 to k-1 do
                Send T to processor j^(b)
                Receive B from processor j^(b)
                If (j_b = 1)
                        S := S + B;
                fi

                T:= T+B;
        endfor
end.
```

Figure 5.5: **Computing the prefix sum in a hypercube**

The prefix sums relative to the whole size W window are obtained as below:

i) If a processor is in the left 2^{t-1} subwindow, then its prefix sum is unchanged.

ii) The prefix sum of a processor in the right subwindow is its prefix sum when considered as a member of a 2^{t-1} window plus the sum of the A values in the left subwindow.

The procedure executed by each procedure j in the hypercube, to compute the prefix sums of A in windows of size 2^k, is shown in Fig.5.5.

Lemma 5.5.3 *The prefix sums of A in windows of size 2^k can be computed in k time steps using the* **Prefix Sum Procedure.**

Proof of Lemma 5.5.3 *The proof of the lemma is again through mathematical induction on the number of iterations executed by the* **Prefix Sum Procedure.** *Let* $l = i_{d-1}, i_{d-2}, \ldots, i_{k+1}, i_k, i_{k-1}, \ldots, i_j, i_{j-1}, \ldots, i_1, i_0$ *be a processor in a d dimensional hypercube.*

For the basis step in our proof, it is clear that after the first iteration of the **Prefix sum procedure,** *the prefix sums in windows of size* 2^1 *are known.*

Let the position of processor l *in the subwindow* $M = i_{d-1}, i_{d-2}, \ldots, i_{k+1}, i_k \ldots, i_{j+1}, i_j$ *be* q. *Then, for the induction step we assume that after* 2^j *iterations the registers* S *and* T *of* l *will contain the sum of the first* q A *registers and the sum of all* A *values in the* 2^j *subwindow, respectively. We have to show that after* 2^{j+1} *iterations, the* S *and* T *registers at* l *will respectively contain the prefix sum and total sum of the* A *values in the* 2^{j+1} *subwindow.*

We note that in the $j+1$ *st iteration* l *will exchange* T *with the processor* l^j. *It will then update its total sum* T, *to be equal to the sum of the original value of* T *and the data* B *it received from its neighbor. Clearly, the new value of* T *is equal to the sum of all* A *values in the* 2^{j+1} *subwindow.*

As for the value of the S *register,* l *will update it to be equal to* B + *the original value of* S, *if the* j^{th} *bit of* $l = 1$, *i.e. if* l *is in the right subwindow of the window* $i_{d-1}, i_{d-2}, \ldots, i_{j+1}$. *Otherwise, there will be no change in the value of register* S *in processor* l. *Thus, after* 2^{j+1} *iterations, each of the* 2^{j+1} S *registers will contain the prefix sums of the* A *values, in the subwindow of size* 2^{j+1}.

It follows, from the principle of mathematical induction, that the **prefix sum procedure** *can compute the prefix sum of a size* 2^k *window, in k time steps.* □

In all of the above procedures, we observe that only the required variables are updated and that there are no side effects.

5.6 Design of the Par_ETF

In section 5.3 we had shown that the complexity of the sequential ETF algorithm is $O(mn^2)$ where m is the number of application processors and n is the number of application tasks.

In this section, we will show how an ETF schedule for the n tasks and m processors can be constructed in parallel. To avoid confusion, we will refer to the tasks and processors that have to be scheduled as application tasks and application processors or simply tasks and processors. The processors that perform the scheduling would be referred to as the scheduling processors.

To obtain maximum parallelism, we assume that the hypercube on which the parallel heuristic would be realized has $O(mn)$ scheduling processors where m and n are some power of 2. These mn processors may be viewed as forming an $m \times n$ array. Hence, we have a scheduling processor at each of the array positions (j, i), $0 \leq j < n$, $0 \leq i < m$. We shall use two different notations for accessing the hypercube processors. One is the usual one dimensional notation. In this the scheduling

processor SP(k) refers to the kth processor in the hypercube, $0 \leq k < mn$. The second is a two dimensional notation. In this SP(j,i) refers to the processor in position (j,i) of our two dimensional view. The mapping between the one and two dimensional notations is done using column major order. If SP(k) and SP (j,i) refer to the same processor then $k = i \times n + j$. Furthermore, if $m = 2^q$ and $n = 2^p$ and the binary representation of k is $l_{q+p-1}, l_{q+p-2}, \cdots, l_0$ then the binary representation of i is $l_{q+p-1}, \cdots l_{p+1}, l_p$; and that of j is $l_{p-1}, \cdots, l_0$.

Also, let $Q(k)$ refer to memory location Q of processor SP(k) and $Q(j,i)$ refer to memory location Q of processor SP(j,i).

In the **parallel ETF scheduling algorithm** each scheduling processor SP(j,i) handles the scheduling of the application task T_j and the processor P_i.

To schedule the application, each scheduling processor SP(k) (= SP (j,i)), $0 \leq k < (mn), 0 \leq j < n, 0 \leq i < m$ in a (mn) processor hypercube executes the procedure shown in Fig. 5.6. It is assumed in the procedure that the each scheduling processor has its own unique I/O environment and the relevant application data. Removing this assumption would alter the time and cost complexities of the algorithm as the distribution of application data is not included.

The parallel ETF algorithm like its sequential counterpart consists of 2 main phases the **initialization phase** and the **scheduling phase**. The scheduling phase may further be divided into five subphases. For the sake of clarity the initialization phase and each of the 5 subphases of the scheduling phase would be dealt

---

Algorithm Par_ETF

Input:

(i) The position of the processor in one and two dimensional notation in the hypercube.

(ii) The total number of application tasks (n) and processors (m).

(iii) The execution time $a(k)$ of task.

(iv) An adjacency array Pred representing the predecessors of the application task.

(v) An adjacency array Succ representing the successors of application task.

(vi) A communication array c representing the number of message exchanges between the application task T_j and its successors.

(vii) An array l representing the time it takes to transfer a unit message from an application processor to any other processor in the application system.

Output

For the application task T_j

i) Its starting time, $s(T_j)$.

ii) Its finishing time $f(T_j)$.

iii) The application processor $p(T_j)$ to which task T_i has been assigned.

Figure 5.6: The Par_ETF algorithm

```
Procedure Par_ETF(k,i,j,m,n,Pred,Succ,c,l,s,f,p)
begin
        Initialize;

        while (Q(k) < n)
                schedule_ready_tasks;
                if (Q(k) < n)
                        calc_new_moment;
                        finish_tasks;
                        calc_ready_times;
                fi

        endwhile
end.
```

Figure 5.6: **Continuation of the Par_ETF algorithm**

separately. It should be noted that each of the procedures mentioned are assumed to be executed by the scheduling processor SP(k) = SP(j,i).

The initialization phase consists of the procedure **Initialize.** This procedure is started by having each of the application processors initialized to free state. This is equivalent to what was done in **SEQ _ETF** where all the application processors were included in the set of available processors. After that the number of predecessors of a task T_j held by a scheduling processor are computed. If the number is equal to zero then T_j is regarded as a **ready task.** The **ready time** r(j, i) of such a task on an application processor i is set to 0. The time complexity of **Initialize** is $O(n)$ using $O(mn)$ scheduling processors.

The next phase of **Par_ETF** is the **scheduling phase.** This phase is repeated until all the tasks have been scheduled. The first step in the phase is to schedule the set of ready tasks which remain unscheduled at every iteration.

```
Procedure Initialize
begin
    Initialize the application processor P_i, held by SP(k) to be free by assigning
    Procstat (k) = free;

    Compute the number of predecessors, NPred(k), of the task T_j
    held by SP(k);

    Let taskstat(k) = ready, if T_j has no predecessors;

    Use the data sum procedure in the window defined by the scheduling
    processors (*,i) to compute the total number of ready tasks,
    taskav(k);

    Q(k) := 0;
    CM(k) := 0;
    NM(k) := ∞ ;
    procav (k) := m;
    if (taskstat (k) = ready)
        r(j,i) = 0;
    fi
end.
```

Figure 5.7: Par_ETF algorithm, Procedure Initialize

We select a task $\hat{T}$ and a processor $\hat{P}$ such that the ready time of $\hat{T}$ on $\hat{P}$ is the minimum of all the ready times. This can be done in $O(\log m + \log n)$ time, by using the procedure **Min** in the window defined by the mn processors in the hypercube. The proposed earliest starting time of the selected task is maximum of the value of the event clock and its ready time.

Once a task has been selected it has to be decided whether it should be scheduled or not. This decision depends on the value of $\hat{e}_s(k)$. Note that each scheduling processor, k, has the same value of $\hat{e}_s(k)$. If $\hat{\mathbf{e}}_\mathbf{s}(\mathbf{k}) \leq \mathrm{NM}(\mathbf{k})$ then the selected task can be scheduled. As a result of the decision, the number of ready tasks and free processors have to be decremented by 1, and the status of the application task $\hat{T}$ and application processor $\hat{P}$ have to be updated to **scheduled** and **busy** respectively. Thus, each scheduling processor SP(*, $\hat{P}$) updates its local variables **procstat** to be busy; **finish** to be equal to the time the application $\hat{P}$ would be busy; and **task** to be equal to the task assigned to the application processor $\hat{P}$. Also, each scheduling processor SP($\hat{T}$, *) updates its local variables **taskstat** to be scheduled; **p** to be equal to the application processor that is going to execute $\hat{T}$ and **f** to be equal to the time $\hat{T}$ will complete. If the selected task $\hat{T}$ will complete before the next moment NM, then all the scheduling processors will update their local values of NM to be equal to the completion time of $\hat{T}$. Otherwise, as in **SEQ _ETF**, the loop for scheduling ready tasks is terminated.

From our earlier discussions we know that **Procedure schedule_ready_tasks** is executed at most $O(2n)$ times where n is the number of application tasks and that each iteration takes $O(\log m + \log n)$ time to execute. Thus in the worst case the

```
Procedure schedule_ready_tasks
begin
        While (procav(k) ≠ 0) and(taskav(k) ≠ 0) do
                select_ready_task(ê_s(k),T̂,P̂);
                if (ê_s(k) ≤ NM(k))
                        schedule_task(T̂, P̂, ê_s(k));
                else
                        exit the loop;
                fi
        endwhile
end
```

Figure 5.8: **Par_ETF algorithm, Procedure schedule_ready_tasks**

schedule_ready_tasks takes $O(2n(\log m + \log n))$ time for $O(2n)$ iterations using $O(mn)$ scheduling processors.

If **schedule_ready _tasks** terminates and not all the tasks have been scheduled then the **Par_ETF** heuristic executes the remaining subphases of the scheduling phase.

It starts by advancing the event clock and calculating the new value of NM by executing the **calc_new_moment procedure** shown in Fig. 5.11.

The new value of NM(k)=NM(j,i) in the above procedure, is equal to the minimum of the finishing times of the scheduled processors which remain busy till the current moment. Since, a window defined by the scheduling processors (j,*) contains all the possible application processors, it suffices to execute the **Min** procedure in the window defined by the processors (j,*). Thus each execution of **calc_new_moment**

Procedure select_ready_task ($\hat{e}_s(k)$,$\hat{T}$, $\hat{P}$)
begin

Use the procedure Min to find amongst all the ready tasks the task $\hat{T}$ and amongst all the free processors the processor $\hat{P}$ which satisfy

$$r(\hat{T},\hat{P}) = \min_{T \text{ is ready}} \min_{P \text{ is free}} r(T,P);$$

Let temp(k)= $r(\hat{T},\hat{P})$;
Let the execution time of $\hat{T}$ be ex(k);
Let $\hat{e}_s(k)$ = max { CM(k), temp(k) };

end.

Figure 5.9: **Par_ETF algorithm, Procedure select_ready_task**

Procedure schedule_task($\hat{T},\hat{P},\hat{e}_s(k)$)
begin

taskav (k) = taskav(k)-1;
Procav(k) = Procav(k)-1;
Q(k) = Q(k)+1;
temp(k)=$\hat{e}_s$(k)+ex(k); if ($\hat{P} = i$)
  procstat (k) = busy;
  finish(k) = $\hat{e}_s(k)$ + ex(k);
  task(k) = $\hat{T}$;
fi
if ($\hat{T} = j$)
  taskstat (k) := scheduled;
  p(k):=$\hat{P}$;
  f(k):= temp(k);
fi
if (NM(k) > temp(k)
  NM(k)=temp(k);
fi

end.

Figure 5.10: **Par_ETF algorithm, Procedure schedule_task**

Procedure calc_new_moment
begin

CM(k) = NM(k);

Use the **Min** procedure in the window defined by
the processors (j,*) to find new NM(k);

end.

Figure 5.11: **Par_ETF algorithm, Procedure calc_new_moment**

takes $O(\log m)$ time steps using $O(mn)$ scheduling processors.

In the **procedure finish_tasks** of the **Par_ETF algorithm** each scheduling processor SP(k)=SP(j,i) checks whether its application processor is completing at the current moment. If application processor i will be completing now (**finish(k) = CM**), then **procstat(k)** will be updated to be free. Using the **Data Sum procedure**, it then computes the total number of available application processors, **newprocav(k)**. Since a hypercube defined by the scheduling processors SP(j,*) contains the status info of all the application processors the **data sum procedure** would be executed in the window defined by the scheduling processors SP(j,*). **Newprocav(k)** is added to **procav(k)** to give the new value of **procav(k)**. If a task T_j is not a ready or scheduled task each scheduling processor SP(k)= SP(j,i) also computes, **NewPredComp(k)**, the number of remaining immediate predecessors of T_j. It performs this by first checking whether it has just completed an immediate predecessor task of T_j. If it has completed such a task, **NewPredComp(k)** is set to 1 else it is set to 0. Then the **data sum procedure** in the window defined by the scheduling processors SP(j,*) is used to find how many other immediate predecessors

```
Procedure finish_tasks
begin
    if (finish(k) = CM)
        procstat(k) = free;
        newprocav(k) = 1;

        if task(k) is a predecessor of T_j
            set NewPredComp(k) to 1;
        else
            set NewPredComp(k) to 0;
        fi

    else
    newprocav(k)=0;
    fi

    Use the Data Sum procedure in the window defined
    by the scheduling processors SP(j,*) to compute the total
    number of predecessors of T_j that were completed at the
    current moment CM and assign the resulting value
    to NewPredComp(k);

    Use the Data Sum procedure in the window defined by the
    scheduling processors SP(j,*) to compute the newly
    available processors, newprocav(k);

    Set procav(k) = procav(k) + newprocav (k);

    If (NPred (k) > 0)
        Decrement the number of uncompleted processors
        NPred(k), of task T_j, by NewPredComp(k);
        if NPred(k)=0
            taskstat(k)=newav;
        fi
    fi

    rank_new_tasks;

end.
```

Figure 5.12: **Par_ETF algorithm, Procedure finish_tasks**

of T_j have just completed. Note that at any moment at most m tasks can complete. After computing, **NewPredComp(k)**, the number of uncompleted predecessors of task T_j, **NPred(k)**, is decremented by **NewPredComp(k)**. If **NPred(k)** is now equal to 0 then T_j is regarded as a new newly available task.

The most time consuming step in **procedure finish_tasks** is computing the value of **NewPredComp(k)** and **newprocav(k)**. These computations take $O(\log m)$ time steps.

As we observed in **finish_tasks**, a number of applications tasks may become newly available after the completion of some tasks. To facilitate the computation of the ready times of these tasks on the application processors, the newly available tasks should be ranked. This is the function of the **rank_new_tasks procedure**. This procedure generates an ordered list of the newly available tasks. The rank, **prefixwt(k)**, of a newly available task T_j in the list, indicates the number of tasks that have a lower index than T_j and are newly available. The list is generated by using the prefix sum procedure in the window defined by the scheduling processors SP (*,i). As a side result of the above procedure, we can also obtain the total number of newly available tasks, **totwt(k)**. The time complexity of **Procedure rank_new_tasks** is $O(\log m)$.

The last step of the scheduling phase is to find for each newly available task its ready time on each one of the m application processors. We implement this step through the **procedure calc_ready_times**. The procedure starts by broadcasting a newly available task T_j from the scheduling processor SP(j,i), $0 \leq i < n$, to all

```
Procedure rank_new_tasks
begin
        If (taskstat(k) = newav)
                prefixwt(k)=1;
        else
                prefixwt(k)=0;
        fi

        Use the prefix sum procedure in the window defined by the
        scheduling processors, SP(*,i), to compute prefixwt(k), the number of newly
        available tasks which have a lower index than T_j;

                                                    Prefixwt(k)=Prefixwt(k)-1;
        Let totwt(k) be the total number of newly available tasks;
end.
```

Figure 5.13: **Par_ETF algorithm, Procedure rank_new_tasks**

other scheduling processors SP(*,i). Each scheduling processor, SP(s)=SP(v,u), in the scheduling system will regard T_j as **newtask(s)**.

The processor SP(s) will check whether, **newtask(s)**, is a successor of task T_v. If it is a successor then it will compute the time the last message for task **newtask(s)** can arrive at processor P_u from task T_v, which was scheduled on processor p(T_v). Let this time be **temp(s)**. Then using the **procedure Max** in the window defined by the scheduling processors SP(*,u), the maximum, **max(s)**, of **temp(*,u)** is computed. This gives us the time the last message for **newtask(s)** can arrive at processor P_u. The above set of operations is performed for every task that has become newly available at the current value of CM.

At the end of this procedure the number of ready tasks is updated to be equal to the sum of the original number of ready tasks and the total number of newly

available tasks.

As for the time complexity of the above procedure, we observe that computing the ready time of a newly available task takes $O(\log n)$ time. Since in the worst case $O(n)$ application tasks can become newly available, the loop in **Procedure calc_ready_times** may be repeated $O(n)$ times. Thus, the complexity of **calc_ready_times** for $O(n)$, iterations is $O(n \log n)$.

Theorem 5.6.1 *The time complexity of* **Par_ETF** *is* $O(n(\log m + \log n))$ *using* $O(mn)$ *scheduling processors where* n *is the number of application tasks and* m *is the number of application processors and* m *and* n *are both some power of 2.*

Proof of Theorem 5.6.1 *It follows from the above discussion.* □

The algorithm can be adapted to run on scheduling systems of size $k < mn$. If $k = k_r * k_c$ where k_r is the number of rows in the hypercube and k_c is the number of columns in the hypercube, then the time complexity of the algorithm would be $O(n(\frac{nm}{k_r k_c} + \frac{m}{k_c} \log k_r + \frac{n}{k_r} \log k_c))$. The algorithm would be cost optimal if $k_r \leq n/\log n$, and $k_c \leq m/\log m$.

Theorem 5.6.2 *For any EDAG task model* $G = (\Gamma, E, a, c)$ *to be scheduled on a system* $S = (m, l)$*, the procedure* **Par_ETF** *and* **Seq_ETF** *produce the same schedule.*

```
Procedure calc_ready_times
begin
    for w := 1 to totwt(k) do

        if ((prefixwt(k) = w-1) and (taskstat(k) = newav))
            newtask(k) = j;

            Broadcast newtask(k) to all the scheduling
            processors SP(*,i);
        else

            Use the Broadcast procedure in the window
            defined by the scheduling processors SP(*,i)
            to find newtask(k);
        fi


        if (newtask(k) is a successor of task Tj)
            Let temp(k) = f(k) + c[newtask(k)](k)*l[p(k)](k);
            Use the procedure Max in the window defined
            by the scheduling processors SP(*,i) to find
            the maximum value of temp, max(k);
            This resulting maximum value is the ready time of
            newtask(k) on an application processor i;
        fi


        If ((prefixwt(k)=w-1) and (taskstat(k)=newav))
            r(j,i)=max(k);
            taskstat(k)=ready;
        fi

    endfor

    taskav(k)=taskav(k)+totwt(k);
end.
```

Figure 5.14: Par_ETF algorithm, calc_ready_times

Proof of Theorem 5.6.2 *To prove the above theorem, we will use mathematical induction on the application tasks to be scheduled. We will show that the kth task scheduled* $0 \leq k \leq n-1$, *by* **Seq_ETF** *and the kth task scheduled by* **Par_ETF** *are the same tasks and that this task is scheduled to start at the same time slot and is assigned to the same processor by the two algorithms.*

For the basis step it is clear that the first task scheduled by **Seq_ETF** *and* **Par_ETF**, *is the same task and that this task is scheduled to start at the same time slot (0) and is assigned to the same application processor* P_0, *by the two algorithms.*

For the induction step, we assume that the first $i-1$, $2 \leq i < n$, *tasks scheduled by* **Seq_ETF** *and* **Par_ETF** *are the same application tasks and that each of these tasks is scheduled to start at the same time and is assigned to the same application processor by the two algorithms. We have to show that the ith task scheduled by the two algorithms is the same and that it is scheduled by both of them, to start at the same time slot and at the same processor.*

Let T_{j_k} *be the* $(i-1)$ *st task scheduled by the two algorithms and* T_{j_l} *be the ith task scheduled by* **Seq_ ETF**. *Suppose the ith task scheduled by* **Par_ETF** *is* T_{j_m}. *We have to show that* T_{j_l} *is equal to* T_{j_m} *and* $p(T_{j_l}) = p(T_{j_m})$ *and* $s(T_{j_l}) = s(T_{j_m})$.

After the scheduling of T_{j_k}, *there are two possibilities of when the decision of scheduling* T_{j_l} *was made by* **Seq_ETF**:

(i) *The decision of scheduling* T_{j_l} *was made immediately after the decision of scheduling* T_{j_k}.

(ii) *The decision of scheduling T_{j_l} was made after the scheduling of T_{j_k} and inbetween the scheduling T_{j_k} and T_{j_l} , $n_i, 1 \leq n_i \leq i-1$, tasks were completed.*

Each case will be considered separately:

a) The first case implies that after the scheduling of task $i-1$, **Seq_ETF** *found that the set of free processors and available tasks were not empty and that the starting time of T_{j_l} on $p(T_j)$ was less than or equal to NM.*

Since the first $i-1$ tasks scheduled by **Par_ETF** *and* **Seq_ETF** *are the same and since these tasks are assigned the same starting times and to the same processors by the two algorithms, it follows that the following statements hold true:*

i) **Par_ETF** *and* **Seq_ETF** *have the same set of free processors and available tasks after the scheduling of task T_{j_k}.*

ii) The values of NM and CM maintained by each scheduling processor in **Par_ETF** *and by the scheduling processor in* **Seq_ETF**, *after the the scheduling of task T_{j_k}, are the same.*

As a consequence, the **Par_ETF** *algorithm should also schedule T_{j_m} immediately after scheduling T_{j_k}. However, it is known that among all the ready tasks and ready processors, the task $\hat{T}$ and processor $\hat{P}$ selected by the ETF heuristic are chosen such that the ready time of $\hat{T}$ on $\hat{P}$ is minimum.*

Clearly we can't have more than one tasks and more than one processor satisfying the above condition. (Remember, if two or more tasks have the same minimum ready times then the task with the lower task-id is considered to be $\hat{T}$ and if a task $\hat{T}$ has the same ready time on two or more processors then the processor with lower processor-id is considered to be $\hat{P}$).
Thus $T_{j_l}=T_{j_m}$, $s(T_{j_l}) = s(T_{j_m})$, *and* $p(T_{j_l}) = p(T_{j_m})$.

b) The second case occurs if after the scheduling of T_{j_k} *and before the scheduling of* T_{j_l}, Seq_ETF *finds either that there are no ready tasks or that there are no ready processors or the ready times of all the the ready tasks on all the ready processors is more than NM. Thus, in this case it should complete one or more tasks. Let the number of tasks completed by* Seq_ETF *be* n_i, $1 \leq n_i \leq i-1$. *As a result of these events a number of processors would become free, a number of tasks would become available, and CM and NM would get updated.*

From the argument given in (a) it follows that Par_ETF *would also have to complete* n_i *application tasks before scheduling* T_{j_m} *and after scheduling* T_{j_k}. *Since the tasks that are completed during this interval by* Par_ETF *and* Seq_ETF *are the same, we find that after the completion of these tasks:*

i) Par_ETF *and* Seq_ETF *have the same set of free processors and available tasks.*

ii) The values of NM and CM maintained by each scheduling processors in Par_ETF *and by the scheduling processor in* Seq_ETF *are the same.*

Thus it follows from the argument given in (a) that $T_{j_l} = T_{j_m}$*, and* $s(T_{j_l}) = s(T_{j_m})$ *and* $p(T_{j_l}) = p(T_{j_m})$.

From (a) and (b) we thus have, that the ith task scheduled by Par_ETF *and* Seq_ETF *are the same and that this task is scheduled to start at the same time slot and is assigned to the same processor by the two algorithms.*

Hence from the principle of mathematical induction the above theorem holds. □

Corollary 5.6.2: *The performance bound of* Par_ETF *and* Seq_ETF *are the same.*

Chapter 6

Implementation of the Par_ETF algorithm

6.1 Introduction

In the previous chapter the parallel ETF algorithm, its objectives and its design were discussed. In this chapter, we are interested in the implementation issues of the algorithm. An implemented version of the algorithm would allow us to show the correctness of the algorithm and to evaluate its performance relative to the sequential algorithm.

We start out this chapter by providing an overview on transputers, Helios and the CDL language which happen to be the environment chosen to carry out the parallel programming. A particular advantage to the above environment is that

it provides a high-level approach to parallel programming, where the programmer defines the program components, and their relative interconnections and allows the operating system to take care of the actual distribution of these components over the available physical resources.

Then in section 6.4, a procedure is given which constructs a *virtual hypercube* on a transputer using the support provided by the selected environment.

We use this virtual hypercube to implement our **Par_ETF algorithms**. Section 6.5 describes the implementation details of the parallel algorithm. Details of the implemented **Seq_ETF** algorithm are also provided for comparison purposes.

6.2 Par_ETF development environment

For the development of the **Par_ETF** algorithm capable of running on a variety of hypercube architectures, it was necessary that the underlying environment incorporates high-level syntactic support for parallelism. From among the resources available, only the CDL programming language available on transputers came close in satisfying the above requirement. In this section, our purpose is to introduce transputers, the Helios operating system and the CDL Programming Language.

6.2.1 Transputer and the Helios operating system

The transputer is a VLSI microcomputer with a processor, memory and four communication links. There are several types of transputer, for example T400 (32-bit word); T200 (16-bit); T800 (32-bit incorporating a floating point unit) and the recently introduced T9000. Par_ETF has been implemented on the T800.

The programmer's model consists of three-register evaluation stack, a workspace pointer and an instruction pointer.A transputer achieves interprocess communication through channels, which are single words of memory. Two processes that wish to communicate rendezvous at a channel and exchange data by copying them from one buffer to another. As this is implemented by the microcode, the cost of copying lies only in the memory accesses for the data and not in instruction fetches. Communication is strictly one-to-one and channels may not be shared by more than one sender or receiver. The interprocessor links are designed to behave exactly like channels and are used with the same instructions.

Helios is a distributed operating system designed to run on a wide range of transputer based architectures. It is an open system architecture in which parts may be added, removed, modified or replaced transparently to suit specific purposes. In many ways Helios is simply a set of conventions for the behavior of programs. It may be thought of as a 'software backplane' providing an infrastructure for processes to locate and communicate with each other.

The Helios system is based on the client-server model. This technique is widely used in modern operating systems. It divides the tasks into two groups: client

tasks and server tasks. Server tasks control access to the system resources (such as discs, screens, databases and keyboards). Client tasks are application programs which access the system resources by sending requests to the appropriate servers.

The Helios implementation of the client server model allows the clients and servers to be situated in different processors. The standard Helios message passing mechanism (the mechanism by which tasks communicate) works regardless of whether the two tasks are in the same processor, and hence share the same address space or in different processors connected (possibly indirectly), by transputer links. The Helios kernel routes the messages through the intermediate processors until the correct destination processor is located. The message is then passed to the correct task within that processor.

The standard Helios system contains a set of servers and a support program called the I/O server, which runs on the host computer. The standard servers include support for pipes and fifos, both of which are used as a means of communication between two tasks. Amongst the other servers provided are a null server and a ram disc server. All servers interpret the same message protocols and any server will respond not only to requests such as open or read but also to requests for information about the server through such commands as ls.

The Host computer runs the Helios I/O server. It makes the host processor behave much as if it were a transputer running Helios. In particular, the I/O server emulates a number of Helios servers running inside the host computer. These servers depend on the facilities offered by the host but on a PC they include the discs /a,

/b, /c, etc., the ports /rs 232 and /centronics, and the /mouse, /clock, /keyboard and so on.

6.3 CDL Programming Language

The CDL (Component Distribution Language) system provides the user with all that is required to create parallel programs. This means it is not necessary for a user to go down to the level of actually sending messages or writing servers.

The Helios view of a parallel program is not the same as that provided by occam. In occam, the user is presented with processes and processors which are connected by links. The number of links and the topology is fixed within the program, and certain processes run on certain processors. Under Helios, a parallel program consists of tasks connected by pipes. Any number of these software pipes my be used between the tasks and any number of tasks may be specified. The topology of the tasks and pipes is not fixed. At load time, Helios assigns tasks to processors automatically with user provided guidelines on how to perform the placement if required. Although the pipe mechanism may at first appear inefficient it is not so. Two tasks communicating along a pipe send messages directly to each other. These messages are normally sent by each side issuing **read** and **write** posix system calls which map directly onto the low level message passing primitives. These calls also validate the handle passed, and they deal with the retry protocol if a message fails to be delivered.

The CDL language provides a mechanism for constructing networks of tasks and

pipes. Any language such as C or FORTRAN can be used to write the programs. Two levels of support are provided for the distributed programs (**Task Force**).

At the lowest level the tasks referred to as **components of a Task Force** are defined explicitly. This enables the user to define the requirements of individual components. The definition includes the specification of streams on which the component communicates. Streams which are common to different components are used for inter-component communication. In this way the structure of a Task Force (with respect to intercommunication) is defined. The first four channels defined are by convention overloaded onto the standard Unix Streams *stdin, stdout, stderr, stddbg.* If any of these channels is not defined then it defaults to the conventional stream. All stream names must be preceded by a mode. The modes:

< Read

> Write

>> Append

<> Read/Write

< | Read from FIFO

> | Write to FIFO

define in what mode a stream is to be opened and are used to check for stream conflicts and prevent the definitions of streams with multiple readers or writers.

Because streams are opened in the mode specified, an invalid access results in a runtime error.

At a higher level the relationship between components of a Task Force can be defined using a command language. This command language includes simple parallel constructors which define that components execute in parallel and also define how components communicate.

6.4 Realizing a hypercube using the CDL script

We assume that the hypercube on which the **Par_ETF** algorithm is going to be realized has the following characteristics:

(i) Each scheduling processor has access to an I/O Port.

(ii) Every processor knows its position in the hypercube (in one dimension notation)

(iii) Processors communicate with each other by passing messages. There is no global data space where all processors share access to the same data.

(iv) Each processor executes the same program on a different set of input data.

The CDL programming language mentioned in the previous section, was found appropriate for modelling such a hypercube architecture on a transputer. We, as part of the **Par_ETF** package have implemented the program **gen**. This program

generates a task force in which the component tasks and the various communication paths between them form a hypercube topology. The program may be invoked as follows:

% gen {no. of components} {code filename} {data filename}

The first runtime argument of gen is the number of components that are to be generated in the hypercube, the second argument is the name of the program to be executed by each processor; and the last argument is the prefix of the name given to the input and output files. Each component will append its position i in the hypercube to the {data filename} to obtain the name of its input file. For the output it is assumed that the parallel program produces output in only one stream. The name of this stream is the concatenation of {data filename} and 'o1' and append o i to {data filename} to obtain the name of its output file.

The output of the gen program generated in the standard output, is the CDL script defining the hypercube topology. From last section, we know that this is nothing but a definition of a **task force**. Thus each component in the task force definition corresponds to a hypercube processor. We refer this processor as a **virtual hypercube processor.** The task force definition consists of two main parts. The first part is the component declaration. Here, for each component in the task force the streams on which the component is to communicate is specified. The first three streams are left unchanged (i.e. they are standard input, standard output, and standard error). The fourth stream (stream No.3, onwards) specify the communication paths between the component and its neighbors in the hypercube. If the hypercube contains 2^d nodes then each component in the taskforce will contain

$2(d+1)+1$ streams including the standard ones. The jth stream (stream No. $(j-1)$) , $j = i*2+1$, of a component k specifies the communication path from component $k^{(i-1)}$ to component k. Similarly, the lth stream (stream no. $(l-1)$), $l = i*2$ of a component k specifies the communication path from component k to component $k^{(i-1)}$.

In the second part of the definition, the task force is declared as a whole. Here, each component j is given its position in the hypercube as an argument, the standard input of the component j is redirected to its input data file.

The CDL script produced for the following invocation of gen is shown in Fig. 6.1.

% gen 8 paretf pardata

6.5 Modelling of the Par_ETF algorithm on transputers

In section 5.6, the **Par_ETF** algorithm was described. There, it was assumed that the number of scheduling processors is equal to the number of application processors × the number of application tasks. However, in realistic conditions this is not always the case.

In the implemented version of the algorithm, this restriction was eased and the number of scheduling processors were assumed to be $O(k_r * k_c)$ where k_r is the number of rows in the scheduling system and k_c is the number of columns in the

```
# CDL Definition of a 8 processor hypercube
# Task component definition

component A0 { code paretf ;
 streams , , , <I s3, >I s0, <I s7, >I s1, <I s14, >I s2; }
 component A1 { code paretf ;
 streams , , , <I s0, >I s3, <I s10, >I s4, <I s17, >I s5; }
component A2 { code paretf ;
 streams , , , <I s9, >I s6, <I s1, >I s7, <I s20, >I s8; }
component A3 { code paretf ;
 streams , , , <I s6, >I s9, <I s4, >I s10, <I s23, >I s11; }
component A4 { code paretf ;
 streams , , , <I s15, >I s12, <I s19, >I s13, <I s2, >I s14; }
component A5 { code paretf ;
 streams , , , <I s12, >I s15, <I s22, >I s16, <I s5, >I s17; }
component A6 { code paretf ;
 streams , , , <I s21, >I s18, <I s13, >I s19, <I s8, >I s20; }
component A7 { code paretf ;
 streams , , , <I s18, >I s21, <I s16, >I s22, <I s11, >I s23; }

#Task Force Definition

(A0 0 <pardat0 >pardato0) ^^(A1 1 <pardat1 ) ^^
(A2 2 <pardat2 ) ^^(A3 3 <pardat3 ) ^^(A4 4 <pardat4 )
^^(A5 5 <pardat5 ) ^^(A6 6 <pardat6 ) ^^(A7 7 <pardat7 )
```

Figure 6.1: A CDL script for a 3 dimensional hypercube

scheduling system.

6.5.1 Implementation of the Seq_ETF algorithm

Before introducing the implemented version of the Par_ETF program, it would be appropriate for reasons of clarity and comparisons to discuss the implementation of the Seq_ETF program.

The Seq_ETF program has been coded in C in the Helios environment on the transputers. The standard input of the program is the description of the application tasks and processors and in the output we have the ETF schedule.

The input of the program consists of:

(i) The number of application processors;

(ii) The number of application tasks;

(iii) The execution times of the tasks;

(iv) An adjacency matrix A representing the precedence relationships between the tasks.

(v) A communication matrix c, representing the no. of message exchanges between the tasks.

(vi) An adjacency matrix, l, representing the time it takes to send a message packet from a processor P_u to another processor P_v.

A typical input of the program is shown in figure 6.2.

The schedule produced for the input is shown in Fig. 6.3. Note that the text in boldface is not part of the input file and is shown for the sake of clarity. The structure of **Seq_ETF program** is similar to the **Seq_ETF algorithm** mentioned in section 5.3

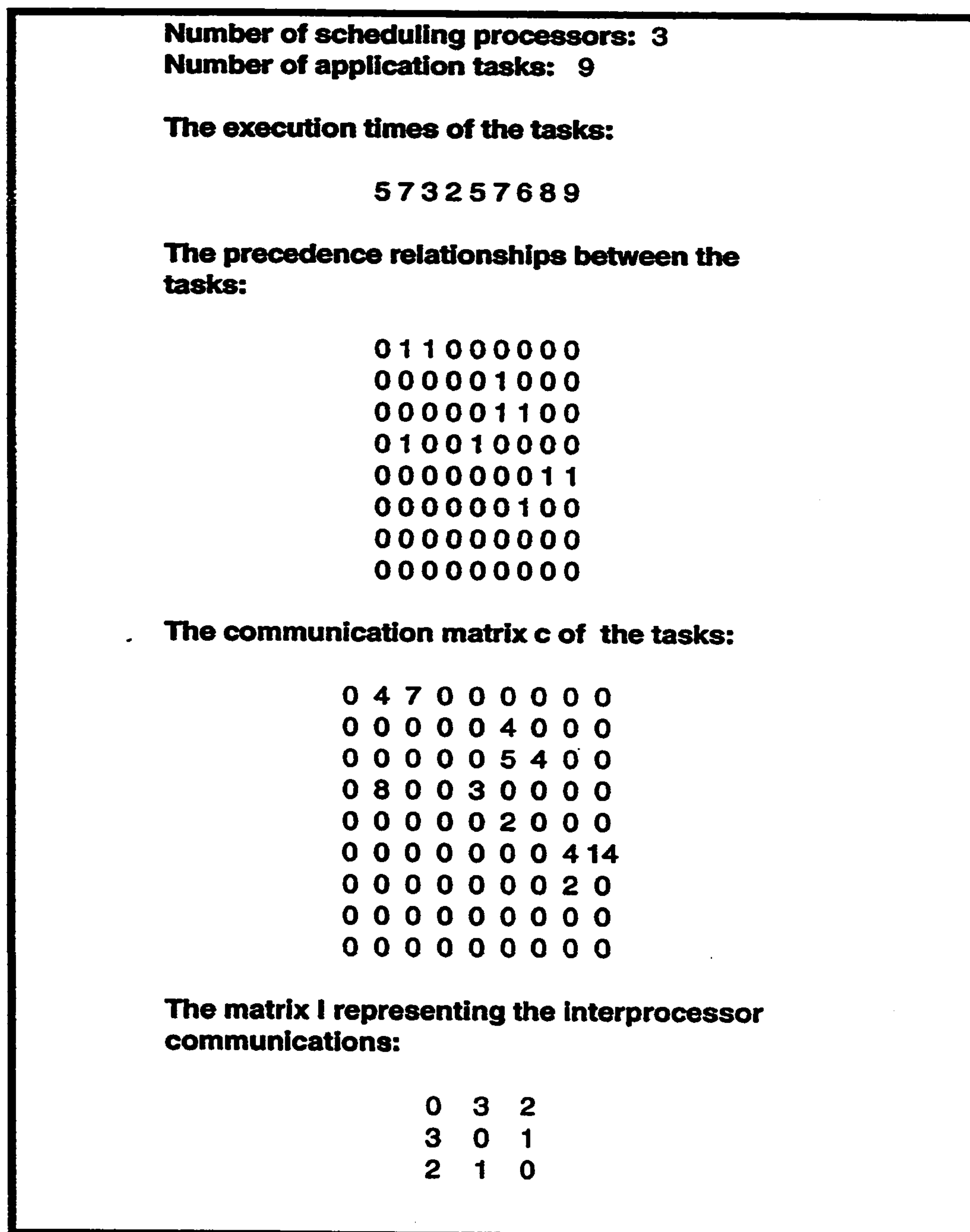
6.5.2 Input of the Par_ETF algorithm

One of the challenging design issues we faced during the development of the **Par_ETF** program was how should each virtual processor acquire its required input. Should it be from the standard input, or the neighboring processors, or from a standalone virtual host processor.

The selected procedure was to have each virtual processor have its own standard input and standard output.

The input consists of all the required information about the application tasks and the application processors, the scheduling processor is responsible for scheduling. Suppose, for example, the number of application tasks is n_1 and the number of application processors is m_1. Also suppose that the required hypercube is arranged as a $n_2 * m_2$ array. Then each scheduling processor SP(j,i) $0 \leq j < n_2$, $0 \leq i < m_1$, is responsible for scheduling the application tasks numbered

(i) from T_{j*n_1/n_2} to $T_{(j+1)*n_1/n_2-1}$, if $((j+1)*n_1/n_2) \leq n_1$);



Number of scheduling processors: 3
Number of application tasks: 9

The execution times of the tasks:

5 7 3 2 5 7 6 8 9

The precedence relationships between the tasks:

```
0 1 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 1 1 0 0
0 1 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

The communication matrix c of the tasks:

```
0 4 7 0 0 0 0 0 0
0 0 0 0 0 4 0 0 0
0 0 0 0 0 5 4 0 0
0 8 0 0 3 0 0 0 0
0 0 0 0 0 2 0 0 0
0 0 0 0 0 0 0 4 14
0 0 0 0 0 0 0 2 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

The matrix I representing the interprocessor communications:

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

Figure 6.2: The input to the Seq_ETF program. The application task and system model are assumed to be same as in Figure 2.1

Task	Processor	starting time	finish time
0	0	0	5
3	1	0	2
4	1	2	7
2	0	5	8
6	0	8	14
1	2	13	20
5	2	20	27
7	2	27	33
8	2	35	44

Figure 6.3: The output of the Seq_ETF for the input shown in Figure 2.1

(ii) from T_{j*n_1/n_2} to T_{n_1-1}, if $((j+1)*n_1/n_2 > n_1)$.

The format of the input file required by each scheduling processor SP(j,i) consists of

(i) The number of application processors (m_1);

(ii) The number of application tasks (n_1);

(iii) The number of scheduling processors in a row of the hypercube (m_2);

(iv) The number of scheduling processors in the column of the hypercube (n_2);

(v) The execution times of the tasks for which it has been assigned.

(vi) An adjacency matrix *Pred* representing the predecessors of the tasks it has been assigned;

(vii) An adjacency matrix *Succ* representing the successors of the tasks it has been assigned.

(viii) A communication matrix, c, representing the number of messages exchanged between the tasks it has been assigned and the successors of these tasks.

(ix) An adjacency matrix, l, representing the time it takes to transfer a unit of message from the processor it has been assigned to all the other application processors.

As part of the **Par_ETF** package, a program **Parconv** has been implemented which will allow the conversion of an input representation of an application task and processor system suitable for the **Seq_ETF** algorithm ,to an input representation appropriate for the **Par_ETF** algorithm. The **Parconv** program may be invoked using the following command structure:

% Parconv {pros in col } {procs in row } {filename}

where { procs in col } specifies the number of processors in a column, and {procs in col } specifies the number of processors in a row. The third parameter is used as the prefix of the names of the input and output files of **Par_ETF** algorithm. The standard input of **parconv** is the input representation of an application task and processor system suitable for the **Seq_ETF** algorithm.

As an example, the following invocation of **Parconv** will produce the input representation of an application task and processor system suitable for a **Par ETF** algorithm running on a $4x2$ hypercube processor system.

% Parconv 2 4 pardat < datafile

The resulting output of the above command would consist of 8 data files starting with the letters 'pardat'. Each resulting data file corresponds to the standard input of a scheduling processor in a 4×2 hypercube processor system. A typical resulting file is shown in Fig. 6.4.

6.5.3 Par_ETF program

Once the application task and processor systems are in the required format, and can be fed to each scheduling processor, the scheduling of these tasks and processors can begin.

The code executed by each scheduling processor is similar to the one described in section 5.6. However, as mentioned earlier in this section we have eased our restriction on the number of scheduling processors.

An important aspect of the program worth mentioning is the method of communication between two neighboring scheduling processors (components). Posix level I/O calls are used for communication between components. As we noted in section 5.6 the **Par_ETF** algorithm is a **normal algorithm** in which communication links are used in only one dimension at a time and that consecutive dimensions are used at consecutive time steps. We also note from the algorithm, that during a communication operation every scheduling processor has both, a data item to send and a data item to receive (even though one or both of these data items may have dummy

No. of application processors: 3
No. of application tasks: 9
No. of scheduling processors in a column : 2
No. of scheduling processors in a row : 4

The execution times of the tasks assigned to SP(4)

5 7 3

The adjacency matrix Pred representing the predecessors of tasks assigned to SP(4):

0 0 0 0 0 0 0 0 0
1 0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0 0

The adjacency matrix Succ representing the successors of tasks assigned to SP(4):

0 1 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 1 1 0 0

The adjacency matrix c representing the communication between the tasks assigned to SP(4) and their successors:

0 4 7 0 0 0 0 0 0
0 0 0 0 0 0 0 4 0
0 0 0 0 0 0 0 5 4

The adjacency matrix l representing the no. of messages sent from the application tasks assigned to SP(4) :

2 1 0

Figure 6.4: Figure showing the input of SP(4) for the Par_ETF program

values).

Thus, in order to synchronize the communication between two neighboring processors i and i^k at the kth dimension, the following scheme is followed by SP(i).

If the kth bit of processor SP (i) is 1

SP (i) writes to the stream $(2*k+2)$ which corresponds to the link from SP(i) to SP(i^k).

SP (i) reads from the stream $(2*k+1)$ which corresponds to the link from SP (i^k) to SP(i).

else

SP(i) reads from the stream $(2*k+1)$ which corresponds to the link from SP(i^k) to SP (i).

SP (i) writes to the stream $(2*k+2)$ which corresponds to the link from SP(i) to SP (i^k).

This ensures that the two-way communication between neighboring processors is deadlock free.

It can be argued that the program is deadlock free with respect to communication. This is due to the usage of the scheme discussed above, and the fact that communication between neighboring processors is being performed in a fixed order. The ordering of the communication requests ensures that a cycle consisting of such requests is not created.

Once the code of each scheduling processor is specified and the task force is defined, an executable version of the task force can be generated by running the CDL compiler on the *CDL script* defining the task force. Running this program would produce an output file containing the scheduling decisions made during the scheduling processes.

Chapter 7

Performance of the Par_ETF Program

7.1 Incorporating the virtual time in the Par_ETF Program

As it may be recalled from Chapter 5, one of the main objectives of the Par _ETF scheduling algorithm is to reduce the time complexity of obtaining an ETF schedule. In order to verify the reduced complexity, it is necessary to know the execution times of the Par_ETF program and the Seq_ETF program. An accurate method of computing the execution time of a parallel or sequential program is to actually measure the time it takes to execute the program from start to finish. This, however, was found to be infeasible for our implementation of the Par_ETF program. This

is because:

1. The number of processors (transputers) available for our implementation is limited (9 to be exact). Thus, it is necessary to simulate execution of a large number of scheduling processors on a small parallel system of transputers. This simulation as mentioned in chapter 6 is performed by having a scheduling task corresponding to each scheduling processor.

2. There is no system routine provided by Helios which can accurately determine, the amount of CPU time used by a task for its computations since its creation.

Because of the above reasons it was necessary to look for other appropriate methods of computing the execution times. The method attributed to Lamport, for ordering events in a distributed system using logical clocks was found to be suitable [37]. The method referred to as **virtual time** can be implemented by ensuring the following:

- A process P_i increments its virtual time V_i by a unit computation cost, between any two successive program statements.

- If a process wants to send a message m and its virtual time is V_i, then the message m bears the time stamp $St_{mi} = V_i$.

- When the message m is received, the virtual time V_j of the receiving processor P_j is compared with the sum of the time stamp St_{mi} of message m and the communication cost between P_i and P_j. If the virtual time V_j of P_j is less

than the sum then the virtual time V_j is set equal to the sum else it remains unchanged. Since the size of a message sent in our program doesn't exceed a constant, the communication cost is taken to be constant. Otherwise, the variation in the message length would have been reflected in the communication cost.

In the **Par_ETF** program we have taken the unit communication cost and computation cost to be equal. It should also be mentioned that the I/O operations are not being timed in the program.

The virtual execution times obtained by executing the **Seq_ETF** and **Par_ETF** program on the system and task graph given in Fig. 2.1 are shown in Table 7.1. The táble shows that in general, as the number of scheduling processors are increased there is a decrease in the execution time of the parallel program. It should also be noted that the execution times of **Par_ETF** also differ if it is executed on the same number of scheduling processors but different scheduling configurations. For example, for the test runs shown in Table 7.1, if we are given 16 processors **Par_ETF** may execute in 1192 , 1193 or 1290 virtual time units depending upon the configuration of the scheduling system.

7.2 Comparisons of Results

In order to evaluate the correctness and the performance of the **Par_ETF** program, several test runs have been conducted. In order to organize the presentation of

scheduling processors	time (virtual units)
1 x 1	2888
1 x 2	2499
2 x 1	1924
4 x 1	1587
4 x 2	1363
4 x 4	1192
8 x 1	1401
8 x 2	1193
8 x 4	1045
16 x 1	1290
16 x 2	1096
32 x 1	1357

Table 7.1: **The execution time of Par_ETF for application system in Figure 1.1**

the results, these tests have been grouped into several classes. The parameters of interest in these classes are the application system configuration, the application task configuration, and the configuration of the scheduling system. In each class we are varying one of the parameters and fixing the other two.

7.2.1 Test class 1: Varying the scheduling system size

Here we are fixing the application system configuration and the application task configuration and varying the no. of scheduling processors. The application system is a hypercube consisting of 4 nodes. The communication cost between two directly connected processors is assumed to be 2 time units. The communication structure between application tasks is also hypercube- like. A task T_i precedes T_j if $i < j$ and i differs from j in exactly 1 bit. We will refer to such task graphs as **hypercube-like**

system size	time (virtual units)
Sequential	7152
1 x 1	9895
2 x 1	5934
4 x 1	4020
8 x 1	3178
16 x 1	2900
1 x 2	7570
2 x 2	4538
4 x 2	3069
8 x 2	2425
16 x 2	2193
1 x 4	6541
2 x 4	3942
8 x 4	2107
16 x 4	1894

Table 7.2: **Execution time of Par_ETF for a fixed application on various scheduling processor configurations**

task graphs. The communication requirement between an immediate processor and an immediate successor tasks is 5 time units. The number of application tasks are 16, and the execution cost of each tasks is a random number between 22 and 80.

The performance of the **Par_ETF** program for the various scheduling system configurations is shown in Table 7.2.

From Table 7.2 it can be observed that the sequential algorithm is faster than the parallel algorithm when the number of scheduling processors available is 1. This is because in the **Par_ETF** algorithm a number of operations are unnecessary when the number of available scheduling processors is only 1.

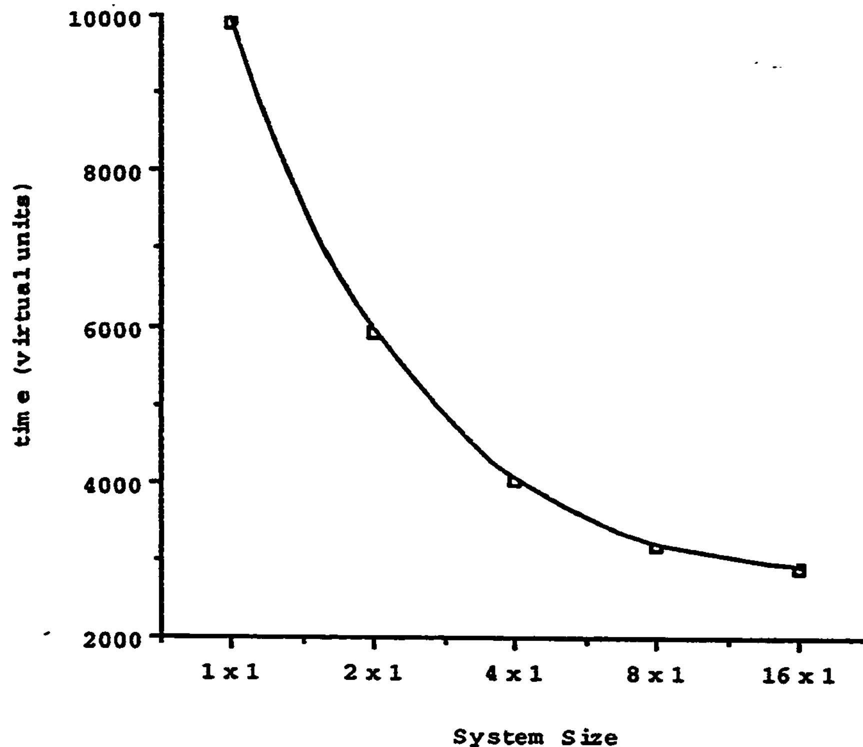


Figure 7.1: **Scheduling completion time vs. the scheduling system size, No. of columns in scheduling system=1**

The 'scheduling completion time vs. the scheduling system size' graphs for some configurations are shown in Fig. 7.1, Fig. 7.2, Fig. 7.3, and Fig. 7.4. The first 3 graphs show the effect of increasing the number of rows in the scheduling system for different number of columns. The fourth graph shows the parallelism obtained as a function of scheduling system size.

Initially, as the number of scheduling processors are increased the speedup of the **Par_ETF** program is almost linear. The speedup is however reduced, as more processors are added to the scheduling system. This confirms to our expectations

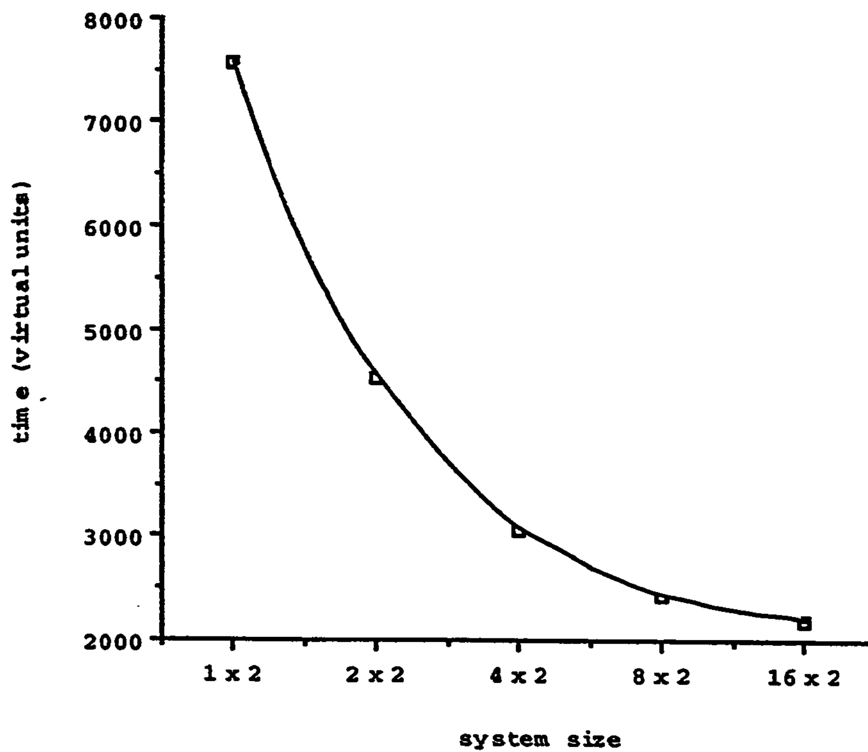


Figure 7.2: Scheduling completion time vs. the scheduling system size, No. of columns in scheduling system=2

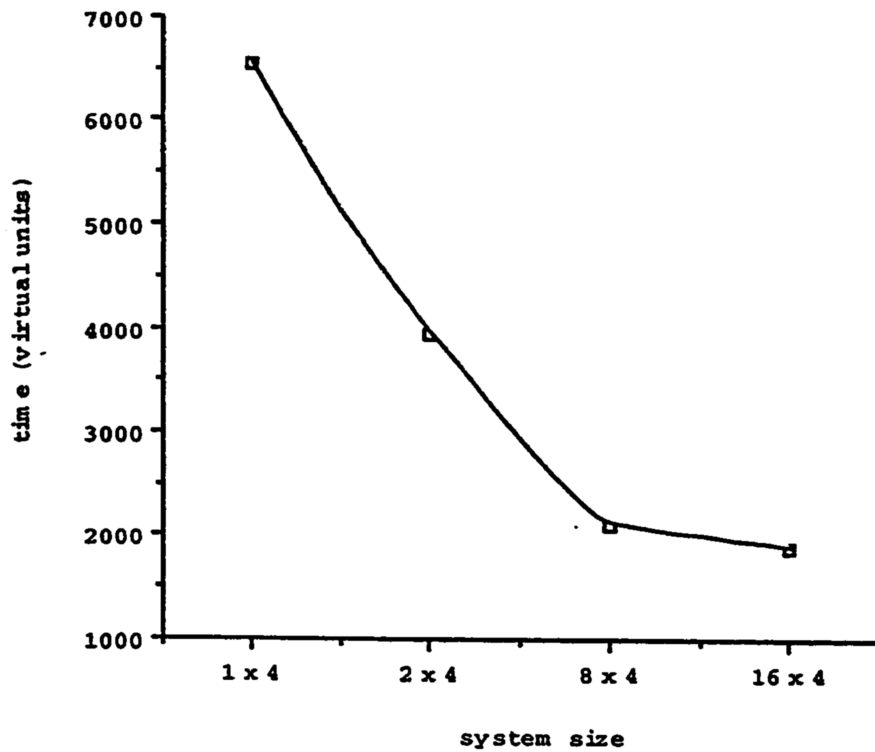


Figure 7.3: Scheduling completion time vs. the scheduling system size, No. of columns in scheduling system=4

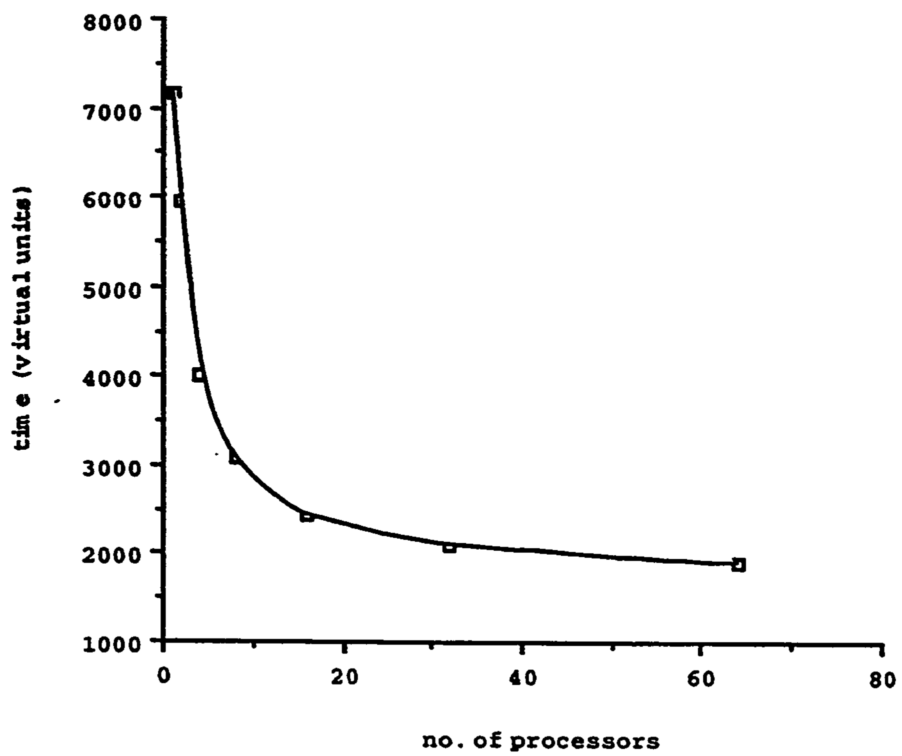


Figure 7.4: Scheduling completion time vs. the no. of scheduling processors

since initially as the the number of scheduling processors are increased the work load of the Par_ETF can be divided among the new processors. However, when sufficient number of processors are available the work can no longer be divided such that the overall time can be reduced. In Fig. 7.4, it can be observed that the decrease in the completion time is not significant after a certain threshold, which is about 16 processors.

7.2.2 Test Class 2: Varying the size of regular structured application systems

Here the application system configuration and the configuration of the scheduling system is fixed, and the varying parameter is the application size. Let the task graph be hypercube-like, with the communication cost between an immediate predecessor and an immediate successor task being 5 time units. The execution cost of the tasks are assumed to be 45 time units. For the experiment the number of tasks are varied from 1 to 64.

Table 7.3 shows the results of three tests conducted with different application sizes for different scheduling system sizes.

In test 2.1 the scheduling system is a 4*2 hypercube and the application system configuration is a 8 processor hypercube. The cost of sending a unit message from a processor to its adjacent processor is 2 time units. In test 2.2 the application system configuration is similar to the one in test 2.1. However, the scheduling system is

task size	time (virtual units)		
	test 2.1	test 2.2	test 2.3
1	71	61	71
2	227	191	227
4	543	452	541
8	1297	882	1301
16	3279	1897	3304
32	10001	5217	9916
64	33326	14803	31521

Table 7.3: **The execution time of Par_ETF for various sizes of hypercube-like task graphs**

test	Sch. system	App. proc. system
test 2.1	4*2 hypercube	8 processor hypercube
test 2.2	8*4 hypercube	8 processor hypercube
test 2.3	4*2 hypercube	8 processor, fully connected

Table 7.4: **Description of Class 2 tests**

changed to a 8*4 hypercube. In test 2.3 the scheduling system is similar to the one in test 2.1. However, the application processor system is changed to a 8 processor fully connected graph. The cost of communication between two adjacent processors is still 2 time units.

The results of the test shown in Table 7.3 are also plotted as shown in Fig. 7.5.

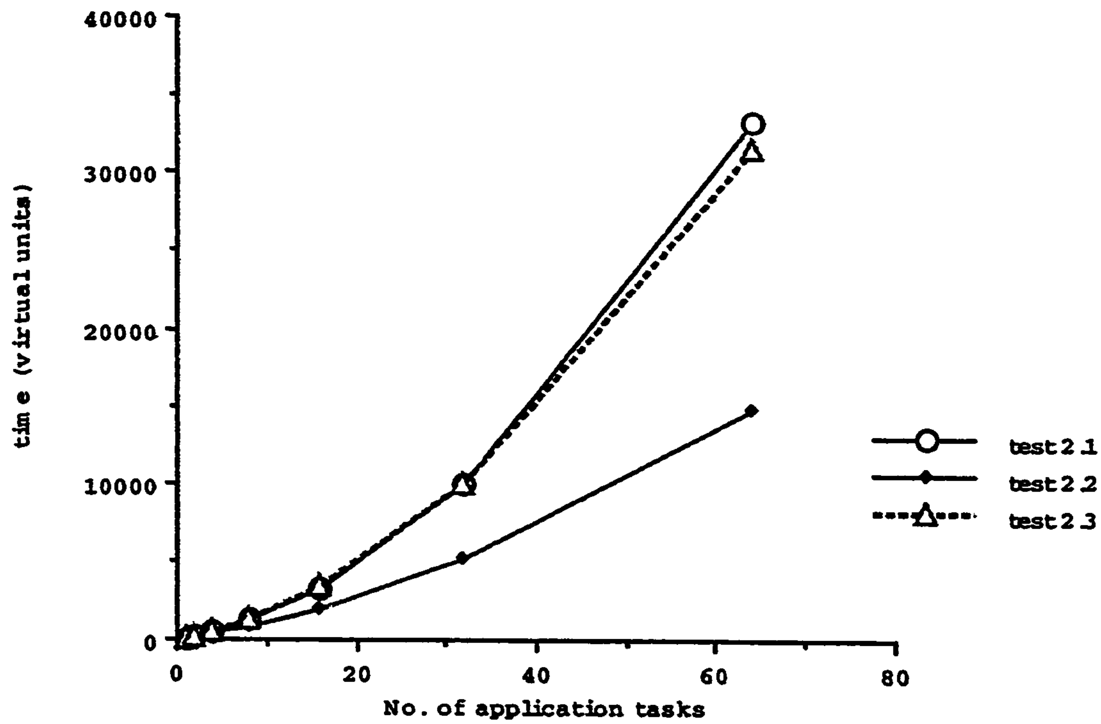


Figure 7.5: Scheduling completion time vs. task graph size. The structure of task graph is hypercube like

Task size	time (virtual units)		
	test 3.1	test 3.2	test 3.3
5	432	307	432
10	1408	968	1406
15	3277	2007	3275
20	5234	3208	5222
25	7997	4681	7925
30	10887	5886	10867
35	14265	7928	14233

Table 7.5: **The execution time of Par_ETF for random task graphs of various sizes**

7.2.3 Test class 3: Varying the task configuration

Here we are fixing the application system configuration and the scheduling system configuration and varying the application task configuration. The precedence relationships between the tasks is random and the size of the task graph ranges from 5 to 35 task nodes. The communication cost between an immediate predecessor and immediate successor is 5 time units.

For tests 3.1, 3.2, 3.3 the application system configuration and scheduling system configuration are similar to the ones in tests 2.1, 2.2, and 2.3.

The results of the test runs are shown in Table 7.5. The task size versus time plot is shown in Fig. 7.6.

From the class 2 and class 3 tests it can be observed that as the size of the tasks graph is increased by a factor of $O(j)$, the increase in the running time of the Par_ETF algorithm is almost $O(j^2)$. This increase is however, not visible for tests

test	Sch. system	App. proc. system
test 3.1	4*2 hypercube	8 processor hypercube
test 3.2	8*4 hypercube	8 processor hypercube
test 3.3	4*2 hypercube	8 processor, fully connected

Table 7.6: Description of Class 3 tests

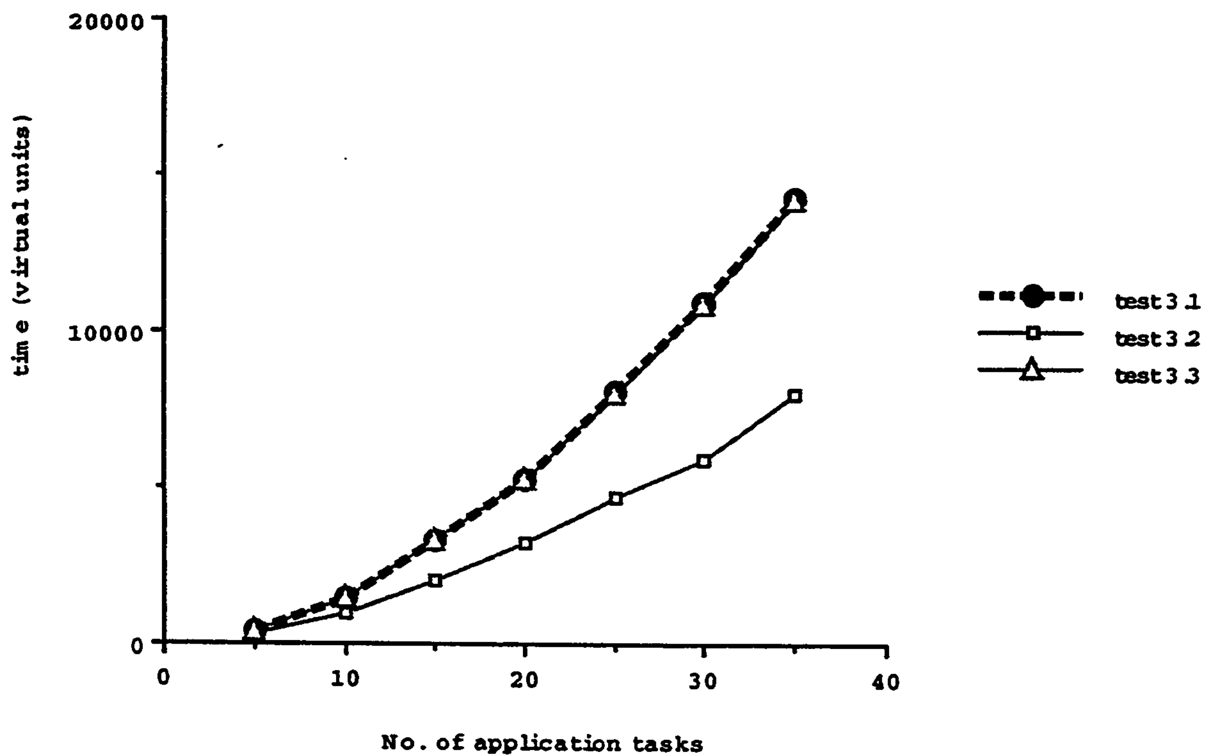


Figure 7.6: Scheduling completion time vs task graph size. Task graph is random in structure

2.2 and test 3.2. This is because the 8*4 scheduling system available in these tests is not being utilized efficiently for small task graphs (see the the discussion for test 1)

From these tests we also note that

- for a large scheduling problem, the 8*4 hypercube scheduling system executes **Par_ETF** faster than a 4*2 hypercube scheduling system. This is intuitive since a 8*4 hypercube scheduling system is able to exploit more parallelism from a large problem than a smaller scheduling system.
- The execution time of **Par_ETF** may vary with the configuration of the application system (although minimally) even if the number of application system processors remain the same. Although this variation is not shown in the theoretical analysis, it can be explained that in general for a given value of the event clock, the average number of scheduling decisions made by the ETF heuristic for a tightly coupled application system is larger than the average number of scheduling decisions made by the ETF heuristic for a loosely coupled system. This is because in a tightly coupled system, an application system can be parallelized more efficiently than in a loosely- coupled system.

7.2.4 Test class 4: Varying the size of random structured application systems

Here the task configuration and the scheduling system configuration are fixed. The varying parameter is the application processor system size. The application processor system is assumed to be a hypercube network with the communication cost between directly connected processors being 2 time units. The network size ranges from 1 to 64.

The application task graph in test 4.1 and test 4.2, is a random task graph with 40 nodes. In test 4.3 the task graph is changed to a 20-node random task graph. For each case, the execution costs of the tasks is assumed to be 45 time units and the communication requirement between an immediate predecessor and immediate successor task is 5 time units. The scheduling system configurations for tests 4.1, 4.2 and 4.3 remains similar to the ones in tests 3.1, 3.2 and 3.3. The results of the test runs are shown in Table 7.7. The application system size versus time graph is shown in Fig. 7.7. From the table and the graph it can clearly be observed that increasing the size of an application by a factor of $O(i)$ would increasing the execution time of the **Par_ETF** program by the same factor. This is in line, with the theoretical complexities derived for the **ETF** algorithms in chapter 5.

From Table 7.7 we also note another interesting behavior of the algorithm. The execution times of **Par_ETF**, running on the scheduling systems of test 4.1, test 4.2 and test 4.3, for 1 application processor is larger than its execution for 2 application processors. We will explain the reason for this for the case of test 4.1. The same

application system size	time (virtual units)		
	test 4.1	test 4.2	test 4.3
1	11452	7843	3517
2	10547	7233	3441
4	12736	7230	3821
8	17435	8831	5234
16	26499	11972	7958
32	44629	18078	13414
64	80917	30299	20579

Table 7.7: **The execution time of Par_ETF for various sizes of application system arranged as hypercubes**

would follow for the other tests.

In test 4.1 the number of scheduling processors in a row is 2 and the number of scheduling processors in a column is 4. When there is only 1 application processor, only the scheduling processors in column 0 are doing useful work (since they are the only one in the system holding the application processor). Thus, actually only half of the scheduling system is being utilized efficiently. We also have the additional overhead of the communication between the processors of column 0 and column 1. This is because in the design of the parallel scheduling algorithm, there is no provision for a scheduling processor to know the statuses of the task and application processor being held by a neighboring scheduling processor, without communication. Thus, a lot of extra and unnecessary operations are being performed here. This doesn't occur when the number of application processors are 2. This is because, now , both columns of the scheduling system can be utilized efficiently.

test	Sch. system	App. task system
test 4.1	4*2 hypercube	40 task nodes, random structured
test 4.2	8*4 hypercube	40 task nodes, random structured
test 4.3	4*2 hypercube	20 task nodes, random structured

Table 7.8: Description of Class 4 tests

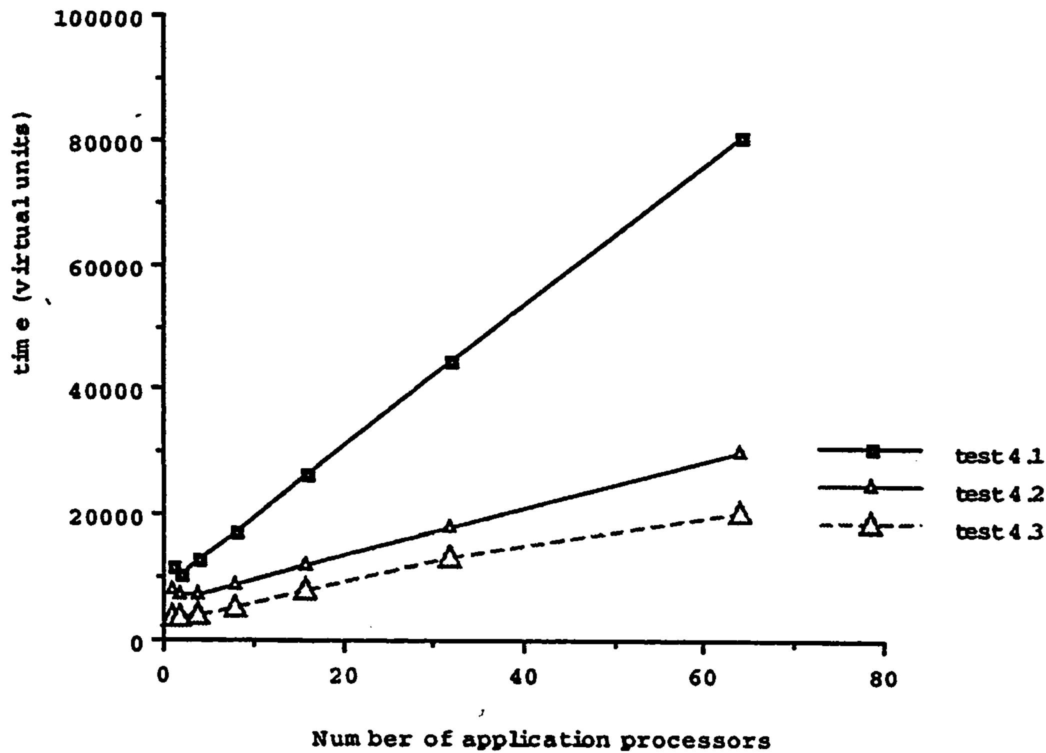


Figure 7.7: Scheduling completion time vs application system size

Chapter 8

Conclusion and Future work

In this thesis, the primary objective has been to design an efficient and fast parallel heuristic for a general task scheduling problem in distributed computing environment. It is assumed that the intertask and interprocessor communication overhead is part of the problem formulation.This approach is found to be feasible as the use of distributed computing systems for numerous parallel and distributed applications is becoming common.

In order to gain an insight to the above problem parallel algorithms for more restrictive cases of scheduling have first been explored. The algorithms explored include the parallel scheduling of tree type task graphs, and UET task graphs.

The parallel task scheduling heuristic designed and implemented in this thesis is based on the **ETF** heuristic [31]. This heuristic referred to as the **Par_ETF** heuristic was designed for the weak hypercube network model. Its running time

was shown to be $O(n(\log m + \log n))$ if $O(mn)$ scheduling processors arranged as hypercubes are used. where n is the number of application tasks and m is the number of application processors. Since the time complexity of the sequential heuristic is also $O(mn^2)$, **Par_ETF** is cost optimal.

As for the schedule produced, it was shown that for any given application **Par_ETF** and **Seq_ETF** produce the same schedule. It, thus, follows that the schedules produced by the two algorithms have the same bounds.

The **Par_ETF** algorithm was implemented on a parallel transputer environment consisting of 9 T800s, running under the Helios operating system. The scheduling system and the interconnections between the processors were specified using the **Component Description Language (CDL)**. Helios and CDL with C provides an efficient parallel programming environment. The facilities found to be lacking in this environment were a suitable debugging tool (both for sequential and parallel programs) and statistical gathering routines.

The results obtained from the implementation of **Par_ETF** using virtual time showed that:.

i) The completion time of the **Seq_ETF** scheduling algorithm is smaller than that of the **Par_ETF** algorithm running on one processor.

ii) The **Par_ETF** algorithm initially speeds up when there is an increase in the number of scheduling processors. However, when there are sufficient scheduling processors relative to the size of the application task graph, and the size

of the application processor graph, an increase in the number of scheduling processors results in little or no change in the execution time of the parallel scheduling algorithm as expected. In fact due to the increased communication requirements the completion may actually increase.

iii) If there is a scheduling problem for which the scheduling system, is being utilized fully, then an increase in the application task graph size by a factor of $O(j)$ would result in an increase in the execution time of the algorithm by a factor of $O(j^2)$.

iv) If there is a scheduling problem for which the scheduling system is being utilized fully, then an increase in the application system size by a factor of $O(j)$ would result in a similar increase in the execution time of the algorithm.

The last two observations show that the algorithm is susceptible more to application size than to the application system size. Many possible extensions exist for the above work. These include:.

i) Design and implementation of **Par_ETF** on other more realistic interconnection networks like a mesh. The results obtained from these implementations could be compared with the results obtained in this work to determine the 'best' suited topology for **Par_ETF**.

ii) Design and implementation of a general parallel scheduling algorithm with a polylog time complexity. This is in contrast to the algorithm implemented as part of this thesis which has a polynomial time complexity.

iii) Design of a parallel scheduling algorithm for restricted task or processor topologies. There the special characteristics of the underlying task or processor topology could be exploited to achieve maximum parallelism.

Bibliography

[1] Agrawal, R., and Jagadish, H.V., "Partitioning techniques for large-grained parellelism," IEEE Trans. on Comput., 37, No.12, pp.1627-1634, Dec. 1988.

[2] Akl,S.G., "The Design and Analysis of Parallel Algorithms," Prentice-Hall, New Jersey, 1989.

[3] Allen, J.R., and Kennedy, K.; PFC; " A program to convert FORTRAN to parallel form," Proc. of the IBM on Parallel Computers and Scientific Computation, Rome, 1982.

[4] Blazewicz, J., "Simple algorithm for multiprocessor scheduling to meet deadlines," Info. Process. Lett., Vol. 6, No. 5, pp.162-174, 1977.

[5] Bokhari,S.H., "A shortest tree Algorithm for optimal Assignments across space and time in a distributed processor system," IEEE Trans. Software Eng., Vol. SE-7, no.6, pp.335-341, Nov. 1981.

[6] Bokhari, S.H., "Partitioning problems in Parallel, Pipelined and Distributed Computing, " IEEE Trans. on Comput., Vol. 37, No. 1, pp.48-57, Jan. 1988.

[7] Brent, R.P., "The parallel evaluation of general arithmetic expression," J. Assoc. Comput. Mach., Vol. 21, pp. 201-206, 1974.

[8] Brucker, P., Garey, M.R., and Johnson, D.S., "Scheduling equal-length tasks under tree like precedence constraints to minimize maximum lateness," Math. Oper. Res., 2, pp. 275-284, 1977

[9] Carlini, U.d., Vilano, U., "Transputers and Parallel Architectures: message paasing distributed systems," Ellis Horwood Ltd., 1991.

[10] Chaudhuri, P., "Parallel Algorithms: Design and Analysis," Prentice Hall, Australia ,1992.

[11] Chou, T.C.K., and Abraham, J.A., "Load balancing in distributed systems," IEEE Trans. Software Eng., Vol.SE-8, No. 4, pp.401-412, July 1982.

[12] Chu, W.W.C., and Lan, L. M-T., " Task Allocation and Precedence Relations for Distrubuted Real Time Systems, " IEEE Trans. on Comput. , Vol. C-36, No. 6, pp.667-677, June 1987.

[13] Coffman, E. Jr., and Graham, C., "Optimal scheduling for two processor systems," Acta informatica, pp. 200-213, 1972.

[14] Coffman,E., "Computer and Job-Shop Scheduling Theory, "Wiley, New York, 1976.

[15] Cole, R., "Parallel Merge Sort, " SIAM J. Comput., Vol.17, No. 4, pp. 770-785, Aug. 1988.

[16] Dekel, E., and Sahni,S., "Binary trees and Parallel Scheduling Algorithms," IEEE Trans. on Comput., Vol. C-32, No.3, pp.307-315, March 1983.

[17] Dolev, D., and Warmuth,M.K., "Scheduling flat graphs," SIAM J. Comput., 14, 3, pp. 638-657, 1985.

[18] Dolev, D., Upfal, E. and, Warmuth, M.K., "The parallel complexity of scheduling with Precedence Constraints," J. Parallel and Distributed Comput.,3, pp. 553-576, 1986.

[19] Efe,K., "Heuristic models of task assignment scheduling in distributed systems," Computer, vol. 15, pp. 50-56, June 1982.

[20] Fenner, P.K., "The Flex/32 for real time multicomputer simulation," in W.J. Korpulus, Ed., Multiprocessors and Array processors, San Diego, CA, Simulation councils Inc., pp. 127-136, Jan. 1987.

[21] Flynn, M.J., "Very high-speed computing systems", Proc. of the IEEE, Vol. 54, pp.1901-1909, 1966.

[22] Fujii, M., Kasami, T., and Ninamiya, K., "Optimal sequencing of two equivalent processors," SIAM J. Appl. Math., 17, pp. 784-789, 1969.

[23] Gabow, H.," An almost linear algorithm for two-processor scheduling," J. Assoc. Comput. Mach., 29, pp.766-780, 1982.

[24] Gabow, H., and Tarjan, R., "A linear time algorithm for special case of disjoint set union," in Proceedings of the 15th Ann. ACM Symp. on Theoryof Computing, Boston, MA, pp.246-251, 1983

[25] Graham, R.L., "Bounds on multiprocessing anomalies," SIAM J. Appl. Math., 17, pp.416-429, 1969.

[26] Hagerup, T., and Rub, C., "Optimal merging and sorting on the EREW PRAM," Info. Process Lett., 33,pp.181-185, Dec. 89.

[27] Helmbold, D., and Mayr, E., " Two Processor scheduling is in NC," SIAM J. Comput., Vol. 16, No. 4, pp.747-759, Aug 1987.

[28] Horn, W.A., "Some simple scheduling algorithms," Naval Res. Logis. Quart., vol. 21, pp.177-185, 1974.

[29] Houstis, C.E., "Module Allocation of Real-Time Applications to Distributed Systems," IEEE Trans. Software Eng., Vol. 16, no. 7, pp.699-709, July 1990.

[30] Hu, T, Parallel Sequencing, and Assembly Line Problems", Operation Research, Vol. 9, pp.841-848, 1961.

[31] Hwang, J-J., Chow, Y.-C., Anger, F.D., and Lee, C.-Y. , "Scheduling precedence graphs in systems with interprocessor communication times," SIAM J. Comput., Vol. 18, No.2, pp.244-257, April 1989.

[32] Hwang, K., "Advanced Parallel Processing with Supercomputer Architectures," Proc. of the IEEE, Vol. 75, No. 10, pp.1348-1379, Oct. 1987.

[33] Jaja, J., "An introduction to Parallel algorithms," Addison-Wesley Publishing Company, 1992.

[34] Kaufman,M.T., "An almost Optimal Algorithm for the Assembly Line Scheduling Problem," IEEE Trans. on Comput., Vol. C-23, No. 11, pp.1169- 1174, Nov. 1974.

[35] Klappholz, D., and Park, H.C., "Parallelized process scheduling for a tightly coupled MIMD machine," 1984 Int. Conf. Proc., Aug. 1984, pp.236-240.

[36] Kruatrachue, B., "Static task sheduling and grain packing in parallel processing systems," Ph.D. thesis, Dept. of Computer Sc., Oregon State Univ., 1987.

[37] Lamport, L., "Time, clocks and the oredering of events in a distributed system, " Journal of the ACM, 33, 2, pp. 327-348, 978

[38] Lee,C.Y., Hwang,J.J., Chow,Y.C., and Ange,F.D., "Multiprocessing Scheduling with Interprocessor Communication Delays," Operations Research Letters, 7, 3, pp.141-147, 1988.

[39] Lo, V.M., "Heuristic Algorithms for Task Assignment in Distributed Systems," IEEE Trans. on Comput., Vol. 37, no. 11, pp. 1384-1397, Nov. 1988.

[40] Lo, Z.-P., and Bavarian, B., "Optimization of Job scheduling on Parallel Machines by Simulated Annealing Algorithms," Expert Systems with Applications, Vol.4, pp.323-328, 1992.

[41] Lundy, M., and, Mees, A., "Covergence of an annealing algorihtm," Math Prog., 34, pp.111-124, 1986.

[42] Ma., P.Y.R., Lee, E.Y.S., and Tsuchiya, " A task allocation model for distributed computing systems, " IEEE Trans. on Comput., Vol. C-31, No. 1, pp.41-47, Jan. 1982.

[43] Nassimi, D., and Sahni, S. "Data broadcasting in SIMD Computers," IEEE Trans. on Comput., vol C-30, pp.101-107, Feb. 1981.

[44] Ousterhout, J. Scelza,D., and Sinhu, P.," Medusa: An experiment in distributed operating system structure," Commun. ACM, vol.23, no.2, pp.92-105, Feb. 1980.

[45] Perihelion Software Ltd, "The Helios Operating System," Prentice Hall, UK, 1989.

[46] Price, C.C., and Salama, M.A., "Scheduling of Precedence-Constrained Tasks on Multiprocessors," The Computer Journal, Vol. 33, No. 3, pp. 219-229, 1990.

[47] El-Rewini,H., and Lewis,T.G., "Scheduling Parallel Program Tasks Onto Arbitray Target Machines," Journal of Parallel and Distributed Computing, June 1990.

[48] Ryu, K.W., "Efficient Parallel Algorithms on the network model," Ph.d. Dissertation, University of Maryland, 1990.

[49] Stankovic, J.A., and Sidhu, I.S., "An adaptive bidding algorithm for processes, clusters and distributed groups," Proc. 4th Int. on Dist. Comp. Systems, pp. 49-59, May 1984.

[50] Stankovic, J.A., "An application of Bayesian decision theory to decentralized control of job scheduling," IEEE Trans. on Comput., Vol C-34, no.2, pp.117-130, Feb. 1985.

[51] Shen, C., and Tsai, W., "A graph matching approach for optimal task assignment in distributed comouting systems using a minimax criterion," IEEE Trans. on Comput., Vol C-34, no.3, pp.197-203, Mar. 1985.

[52] Sinclair, J.B., "Efficient Computation of Optimal Assignments for Distrubited Tasks," J. Parallel and Distributed Comput., 4, pp. 342- 362, 1987.

[53] Smith, V.J., "UET scheduling with unit interprocessor communication delays," Disc.Appl. Math., 18, pp. 55-71, 1987.

[54] Snir, M., "On parallel searching," SIAM J. Comput., 14, pp.688-708, 1985.

[55] Stone, H.S., "Multiprocessor scheduling with the aid of Network flow algorithms," IEEE Trans. Software Eng., No.3, pp. 85-93, March 1977.

[56] Vazirani, U., and Vazirani, V.," The two-processor scheduling is in RNC," in Proceedings of the 17th Ann. ACM Symp. on Theory of Computing, Providence, RI, pp.11-21,1985.

Vita

- Amin Arshad AbdulGhani
- Born at Karachi, Pakistan
- Received Bachelor's degree in Computer Science from King Abdul Aziz University, Jeddah, Saudi Arabia in January 1991.
- Completed Master's degree requirements at King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia in April 1993.