Spring 1995

Grain-size optimization and scheduling for distributed memory architectures

Jing-Chiou Liou
New Jersey Institute of Technology

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

GRAIN-SIZE OPTIMIZATION AND SCHEDULING FOR DISTRIBUTED MEMORY ARCHITECTURES

by
Jing-Chiou Liou

The problem of scheduling parallel programs for execution on distributed memory parallel architectures has become the subject of intense research in recent years. Because of the high inter-processor communication overhead in existing parallel machines, a crucial step in scheduling is task clustering, the process of coalescing heavily communicating fine grain tasks into coarser ones in order to reduce the communication overhead so that the overall execution time is minimized.

The thesis of this research is that the task of *exposing* the parallelism in a given application should be left to the algorithm designer. On the other hand, the task of *limiting* the parallelism in a chosen parallel algorithm is best handled by the compiler or operating system for the target parallel machine. Toward this end, we have developed CASS (for Clustering And Scheduling System), a task management system that provides facilities for automatic granularity optimization and task scheduling of parallel programs on distributed memory parallel architectures.

In CASS, a task graph generated by a profiler is used by the clustering module to find the best granularity at which to execute the program so that the overall execution time is minimized. The scheduling module maps the clusters onto a fixed number of processors and determines the order of execution of tasks in each processor. The output of scheduling module is then used by a code generator to generate machine instructions.

CASS employs two efficient heuristic algorithms for clustering *static* task graphs: CASS-I for clustering with task duplication, and CASS-II for clustering without task duplication. It is shown that the clustering algorithms used by CASS

outperform the best known algorithms reported in the literature. For the scheduling module in CASS, a heuristic algorithm based on *load balancing* is used to merge clusters such that the number of clusters matches the number of available physical processors.

We also investigate task clustering algorithms for *dynamic* task graphs and show that it is inherently more difficult than the static case.

GRAIN-SIZE OPTIMIZATION AND SCHEDULING FOR DISTRIBUTED MEMORY ARCHITECTURES

by
Jing-Chiou Liou

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Electrical and Computer Engineering

May 1995

APPROVAL PAGE

GRAIN-SIZE OPTIMIZATION AND SCHEDULING FOR DISTRIBUTED MEMORY ARCHITECTURES

Jing-Chiou Liou

Dr. Michael A. Palis, Dissertation Advisor — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. John Carpinelli, Committee Member — Date
Associate Professor of Electrical and Computer Engineering
and Director of Computer Engineering
and Acting Associate Chairperson of Electrical
and Computer Engineering, NJIT

Dr. Edwin Hou, Committee Member — Date
Assistant Professor of Electrical and Computer Engineering, NJIT

Dr. David Nassimi, Committee Member — Date
Associate Professor of Computer and Information Science, NJIT

Dr. David Wei, Committee Member — Date
Associate Professor of School of Computer Science
and Engineering, University of Aizu, Japan

BIOGRAPHICAL SKETCH

Author: Jing-Chiou Liou

Degree: Doctor of Philosophy

Date: May 1995

Undergraduate and Graduate Education:

- Doctor of Philosophy in Electrical Engineering,
  New Jersey Institute of Technology,
  Newark, New Jersey, 1995
- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology,
  Newark, New Jersey, 1993
- Bachelor of Science in Electronic Engineering,
  National Taiwan Institute of Technology,
  Taipei, Taiwan, ROC, 1983

Major: Electrical Engineering

Presentations and Publications:

M. A. Palis, J.-C. Liou and D. S. Wei, "A Greedy Task Clustering Heuristic That is Provably Good." Proc. 1994 International Symposium on Parallel Architectures, Algorithms and Networks, Kanazawa, Japan, December 1994.

M. A. Palis, J.-C. Liou and D. S. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures." To appear in IEEE Transactions on Parallel and Distributed Systems.

M. A. Palis, J.-C. Liou, S. Rajasekaran, S. Shende and D. S. Wei, "Online Scheduling of Dynamic Trees." Submitted to Parallel Processing Letters. Special Issue on Partition and Scheduling for Distributed Memory Architectures.

This dissertation is dedicated to
my wife and my parents

ACKNOWLEDGMENT

I wish to express my sincere gratitude to my advisor, Professor Michael A. Palis, for his guidance, friendship, and always being there for me throughout this research. His spiritual advice had a major impact on my development as a researcher and as an individual. I would also like to thank the other members of my dissertation committee Professors John Carpinelli, Edwin Hou, and David Nassimi for their time, support and valuable comments.

Special thanks to Professor David Wei for his being in my dissertation committee and also for helping me in various ways during my research, and Professors Sotirios Ziavras and Edwin Cohen for their help during my graduate work.

I also like to thank my friends who helped me in different ways during my stay at New Jersey Institute of Technology: Zhijian Zhu, Adrienne Walker, Anna Thomas and many others.

And finally, a deep appreciation goes to my wife, Su-Chiou Tsay, for her love and support. She provided various comments on my thesis work, shared my typing load and helped me in analyzing the experimental results.

This work was partially support by New Jersey Institute of Technology Graduate Fellowship, by NSF Grant IRI-9296249, by SBR Grants 421690 and 211665 from the New Jersey Institute of Technology, and by a Group Research Development Grant from the University of Aizu, Japan.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1
INTRODUCTION

1.1 Thesis Motivation

In the last decade, massively parallel processing (MPP) has become consensus approach to high-performance computing. MPP vendors have leveraged the small size, low cost, and high performance of commodity microprocessors to build large-scale parallel machines with hundreds or even thousands of nodes. These powerful machines are now capable of performing billions of floating-point operations per second (gigaflops) and are expected to reach the teraflop level (1000 gigaflops) by the year 2000.

Although the peak performance of MPP machines are impressive, they are rarely achieved in practice. A typical application program running on an MPP machine distributes its tasks and data among the processing nodes and relies on *message-passing* to transfer data between tasks or to synchronize the tasks operations. At the physical level, the resultant inter-node communication causes some nodes to sit idle waiting for data. In existing MPP machines, this communication overhead can be large, typically in excess of 500 instruction cycles [11]. As a result, the actual performance of an application often falls short of its theoretical performance, except for a few "embarrassingly parallel" applications that do not require inter-task communication.

A parallel program can be viewed abstractly as a collection of tasks, where each task consists of a sequence of instructions and input and output parameters. A task starts execution only after all of its input parameters are available; output parameters are sent out to other tasks only after task completion. This notion of a task is called the "macro-dataflow model" by Sarkar [43] and is used by other researchers

[42, 16, 27, 40, 49, 51]. Loosely speaking, the *granularity* (or *grain size*) of a task is the ratio of its execution time vs. the overhead incurred when communicating with other tasks. The granularity of a parallel program is the minimum granularity of its constituent tasks. (A more precise definition of granularity will be given later.)

The high communication overhead in existing MPP machines imposes a minimum threshold on task granularity below which performance degrades significantly. Consequently, to obtain maximum performance, a fine-grain parallel program (i.e., a program with small granularity) may have to be restructured to produce an equivalent coarse-grain program by coalescing many fine-grain tasks into a single task. Manual "fine-tuning" of a parallel program is, unfortunately, too excessive a burden to place on the shoulders of an algorithm designer, be (s)he novice or expert. Not only does (s)he have to deal with the difficult problem of *exposing* the parallelism in a given application, but (s)he also needs to worry about the equally difficult problem of *limiting* the parallelism in the algorithm to minimize communication overhead. Moreover, the latter problem requires of the designer deep knowledge of the characteristics of the target MPP machine, e.g., the number of processing nodes, CPU speed, local memory size, message transfer rate, and network topology. Finally, the fine-tuned program, while it would run with maximum performance on the chosen machine, would in all likelihood perform poorly on another machine with different architectural characteristics. The designer would have to rewrite the program to tune it to the new machine.

1.2 Research Goals

The thesis of this research is that the task of *exposing* the parallelism in a given application should be left to the *algorithm designer*, who has intimate knowledge of the application characteristics. On the other hand, the task of *limiting* the parallelism

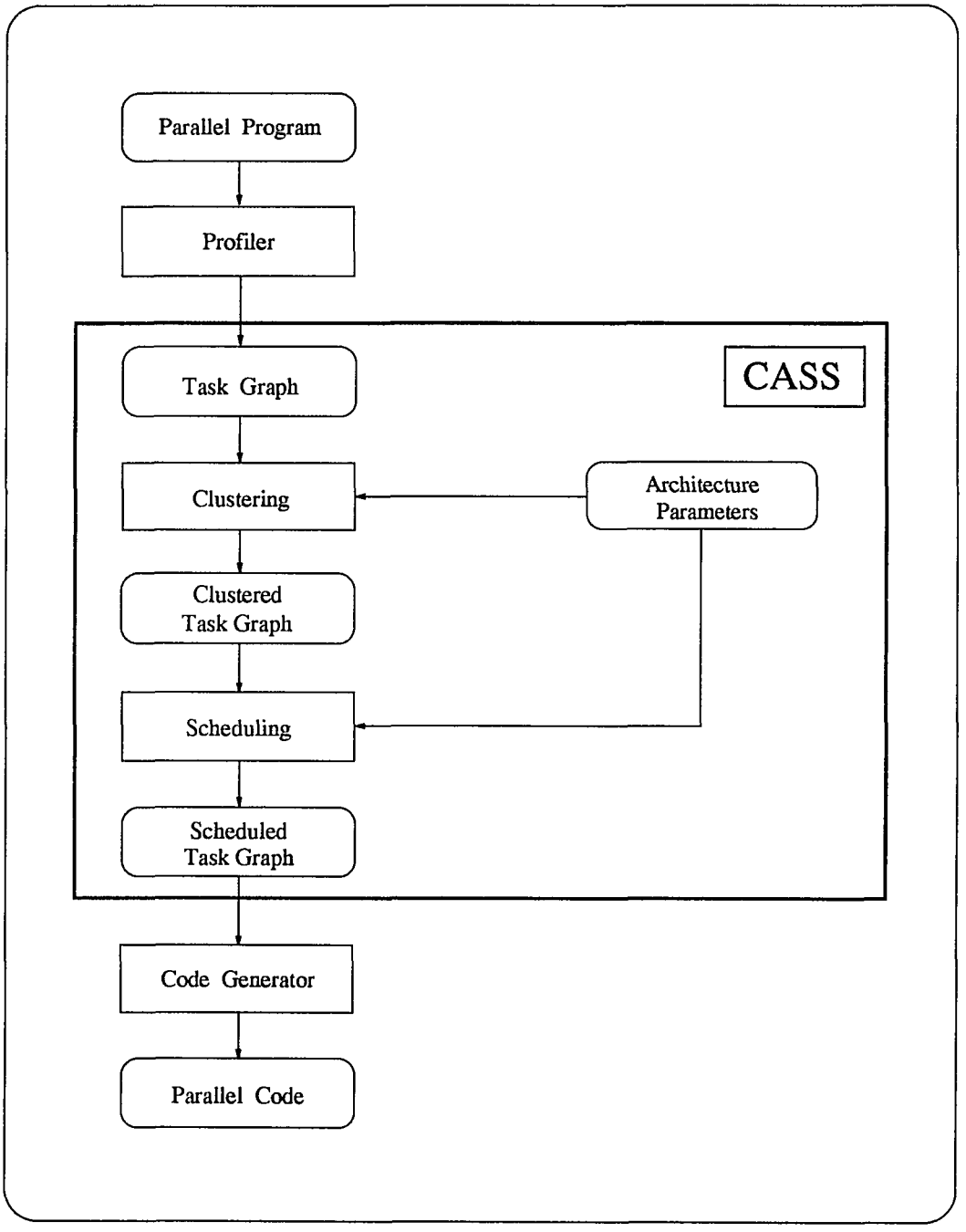


Figure 1.1 The functional modules of CASS.

in a chosen parallel algorithm is best handled by the *compiler* or *operating system* for the target MPP machine. Toward this end, we have developed *CASS* (for *C*lustering *A*nd *S*cheduling *S*ystem), a task management system that provides facilities for automatic granularity optimization and task scheduling of parallel programs on distributed memory parallel architectures.

The main functional modules of *CASS* are shown in Figure 1.1. Given a parallel program and a target parallel machine, a profiler generates a task graph specifying the dependencies among the tasks of the program, the task execution times and the inter-task communication delays. The task graph is used by the clustering module to find the best granularity at which to execute the program so that the overall execution time is minimized. The output of the clustering module is a clustered task graph, in which each cluster represents a collection of tasks that are to be mapped to the same processor. The scheduling module maps the clusters onto a fixed number of processors and determines the order of execution of tasks in each processor. The output of the scheduling module is then used by the code generator to generate machine instructions and to insert communication and synchronization primitives at appropriate points in the generated code.

The clustering module identifies the optimal number of processing nodes that the program will require to obtain maximum performance on the target parallel machine. Our approach is to decouple the clustering algorithm from the scheduling algorithm that actually maps the clusters to the physical processors. There are several reasons for adopting this approach. Firstly, it facilitates *scalability analysis* of the parallel program, i.e., how program performance is affected as the number of physical processors is increased or decreased. In *CASS*, the user may specify *a prior* desired number of processors on which the program is to be run, and a compile-time scheduler will generate code for the appropriate number of processors by *merging clusters*. (Of course, if the number of processors is more than the number of clusters,

no cluster merging is needed.) This two-phase method - task clustering and cluster merging - is more efficient than the one-phase method that performs partitioning and scheduling in the same algorithm. In the former, re-scaling the program so that it runs in a new number of physical processors only requires re-running the cluster merging step, which examines a smaller data set (the clusters), while the latter requires re-executing the entire partitioning and scheduling algorithm on the original fine-grain task graph.

Another motivation for adopting the two-phase method is that in multiprogramming environments where the physical machine is shared by many users, the number of available processors may not be known till run time. In general, a run-time scheduler incurs significant scheduling overhead, proportional to the number of scheduled tasks, that can degrade the performance of the parallel program. The advantage of our approach is that task clustering dramatically reduces the number of tasks to be scheduled at run time, thereby minimizing the effect of scheduling overhead on program performance.

1.3 Originality and Significance of the Research

Previous work on compilers for parallel machines have focused largely on "parallelizing" or "vectorizing" compilers that automatically extract parallelism from existing sequential programs (e.g., "dusty-deck" FORTRAN programs). While such compilers have their niche of applications, there is a greater and more pressing need to develop compilers for parallel programming languages that incorporate language constructs for explicitly expressing concurrency in programs.

Many existing compilers for parallel programming languages do not perform granularity optimization, and if at all, make use of very simple algorithms. For example, the Connection Machine compiler maps "virtual processors" (the

processors used by the program) to physical processors by distributing the virtual processors equally among the physical processors without regard to communication usage patterns. In some compilers, more sophisticated techniques for granularity optimization are used, but they can only be applied to certain segments of the parallel code. For example, loop spreading and loop coalescing are commonly used for granularity optimization: loop spreading distributes iterations of a FOR loop across different processors, while loop coalescing combines several loops into a single loop for execution by the same processor. However, these techniques are only applicable at the loop level and can not be used to optimize the granularity of program segments that exist within loops or are outside of loops.

Our research addresses the granularity optimization problem in a more general context by using a parallel program representation (the task graph) that is essentially language independent. Moreover, unlike previous work which focuses on optimization for *specific* architectures (as is the case for most commercial compilers), our investigation uses a parameterized parallel machine model, thus allowing us to develop granularity optimization techniques that are applicable to a wide range of parallel architectures. Consequently, we will not only be able to assess the effectiveness of today's parallel machines in solving MPP applications, but we will also be able to determine the key architectural features required by these applications, whether these features exist in current machines, and how future MPP machines should be built in order to solve MPP applications much more efficiently and cost-effectively.

1.4 Summary of Research Contributions

1.4.1 CASS Clustering Module

Finding a clustering of a task graph that results in minimum overall execution time is an NP-hard problem [7, 40, 43]. Consequently, practical algorithms must

sacrifice optimality for the sake of efficiency. We have investigated two versions of the problem: clustering without task duplication and clustering with task duplication. In clustering without task duplication, the tasks are partitioned into disjoint clusters and exactly one copy of each task is scheduled. In clustering with task duplication, a task may have several copies in different clusters, each of which is independently scheduled. In general, clustering with task duplication produces shorter schedules than the ones produced by clustering without task duplication. We have developed efficient heuristic algorithms for these two problems, established theoretical bounds on the quality of solutions they generate, and validated the theoretical performance empirically.

When task duplication is allowed, we have an algorithm (CASS-I) which for a task graph with arbitrary granularity, produces a schedule whose makespan is at most twice optimal. Indeed, the quality of the schedule improves as the granularity of the task graph becomes larger. For example, if the granularity is at least $\frac{1}{2}$, the makespan of the schedule is at most $\frac{5}{3}$ times optimal. For a task graph with $|V|$ tasks and $|E|$ inter-task communication constraints, CASS-I runs in $O(|V|(|V|lg|V|+|E|))$ time, which is $|V|$ times faster than the current known algorithm for this problem [40].

We have validated the performance of CASS-I experimentally. Our empirical results demonstrate that CASS-I outperforms the currently best known algorithm in terms of both speed and solution quality.

We have also shown that CASS-I can be used to solve the clustering problem for tree-structured task graphs with *no* task duplication. The algorithm produces a schedule whose makespan is at most twice optimal. This result is interesting because it is known that clustering without task duplication remains NP-hard even when restricted to trees [7].

Unfortunately, we are unable to find a *provably* good clustering algorithm with *no* task duplication for general task graphs. This problem appears to be very difficult because it is known that for general task graphs, clustering with no task duplication remains NP-hard even when the solution quality is relaxed to be within twice the optimal solution [40]. Consequently, we directed our efforts to develop an algorithm (CASS-II) which has fast time complexity of $O(|E| + |V|lg|V|)$ and good *empirical* performance.

We compared CASS-II with the DSC algorithm of [18], which is empirically the best known algorithm for clustering without task duplication. Our experimental results indicate that CASS-II outperforms DSC in terms of speed (3 to 5 times faster). Moreover, in terms of solution quality, CASS-II is very competitive: it is better than DSC for grain sizes less or equal to 0.6, and its superiority increases as the DAG becomes increasingly fine grain. On the other hand, for task graph with grain size 0.6 or greater, DSC becomes competitive and in some cases even outperforms CASS-II, but by no more than 3%.

1.4.2 CASS Scheduling Module

The scheduling module maps the task clusters produced by the clustering module onto a fixed number of processors. If the number of task clusters is greater than the number of processors, a clustering merging step is performed. We investigated three approaches for cluster merging. *Load Balancing* maps the clusters onto processors so that the processors have equal workload (i.e., sum of task execution times). *Communication Traffic Minimizing* maps the clusters onto processors so that the total amount of inter-processor communication is minimized. Finally, *Random* maps the clusters onto processors in a random fashion.

Our experimental results show that when task clustering is performed prior to scheduling, load balancing (LB) is the preferred approach for cluster merging.

LB is fast, easy to implement, and produces significantly better final schedules than Communication Traffic Minimizing (CTM). While CTM outperforms LB for fine grain task graphs, such a situation never arises in the two-phase method of CASS because the task clustering phase produces coarse grain task graphs, for which LB is clearly superior to CTM.

We have also compared the two-phase method with the one-phase method of scheduling. In the one-phase method, the number of the physical processors is used as one of the parameters and those three approaches for cluster merging are applied to task graphs directly. On the contrary, in the two-phase method (which is used by CASS), the clustering module determines the best clustering for the task graph according to its granularity, and the scheduling module matches the number of the clusters to the number of the physical processors. The empirical results show that the two-phase method is superior to the one-phase method in terms of both speed and solution quality, regardless of the cluster merging heuristic used. Indeed, our experimental results indicate that it is not necessary to utilize all processors in the system to obtain a "good" schedule. In other words, the clustering module in the two-phase method can find a near-optimal clustering whose the number of the clusters is less than the number of the physical processors, then the utilization of all processors in the one-phase method may produce a schedule worse than the previous one.

1.4.3 Dynamic Task Graphs

CASS-I and CASS-II are applicable only to parallel programs that can be characterized by *static* task graphs. Such a program consists of a fixed collection of tasks whose execution times and inter-task dependencies and communication delays can be estimated at compile time. On the other hand, many parallel programs give rise to *dynamic* task graphs. In such programs, the number of tasks changes dynamically at runtime. A currently executing task can spawn new tasks which in turn

can communicate with other executing tasks and spawn other tasks. Therefore, the task execution times and inter-task dependencies cannot be known in advance. Programs with dynamic task graphs arise in a variety of applications such as particle simulations, adaptive multigrid algorithms, n-body simulations, and combinational optimization [15].

Because the complete task graph is not known in advance, clustering algorithms based on *global critical path analysis* are not applicable. This method relies on global information about the longest path in the task graph to determine the clustering. It is the method employed by most algorithms for scheduling static task graphs, including our CASS-I and CASS-II algorithms.

For dynamic task graphs, clustering must be performed *online*; i.e., scheduling decisions must be made solely on the basis of the portion of the task graph revealed so far, and not on future tasks. The *competitive ratio* is used for the performance analysis of online scheduling algorithms. Roughly speaking, the competitive ratio is the ratio between the makespan of the schedule produced by an optimal $offline$ scheduling algorithm (that knows the entire task graph in advance) and the makespan of the schedule generated by an online scheduling algorithm. In this research, we have found a lower bound on competitive ratio of any randomized dynamic tree scheduling and a deterministic online algorithm that matches the bound. We show that any online tree scheduling algorithm, even a randomized one, has competitive ratio $\Omega((\frac{1}{g})/log_d(\frac{1}{g}))$ for trees with granularity at most $g < 1$ and degree d.

1.5 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we address the impact of program partitioning and granularity on scheduling for parallel architectures, and also discuss some existing clustering and scheduling approaches. In Chapter 3, a fast

static clustering algorithm (CASS-I) allowing task duplication is presented, and the performance bounds and complexity of CASS-I are shown. In Chapter 4, we present another static clustering algorithm that allows no task duplication. In Chapter 5, we present performance comparison and experimental results for task clustering algorithms to show that the clustering algorithms used by CASS outperform the best known algorithms reported in the literature. In Chapter 6, we describe algorithms for mapping clustered tasks onto physical processors, and present some experiments to show that a load balancing heuristic outperforms the other algorithms. In Chapter 7, we extend our scope from static task graphs to dynamic ones. We adopt a framework of analyzing online scheduling algorithms and based on that we derive a lower bound and a deterministic algorithm that matches that bound. In Chapter 8, we summarize the research work and discuss future work.

CHAPTER 2
BACKGROUND

2.1 Parallel Architectures

A parallel architecture is a computer system with two or more processors connected by an interconnection network as shown in Figure 2.1.

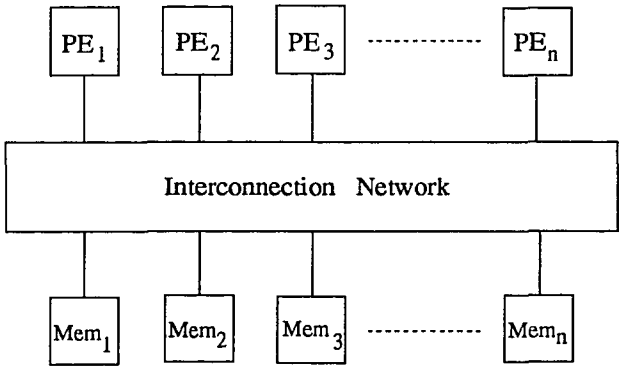


Figure 2.1 A parallel architecture with n processors and n memory units.

Most modern parallel architectures can be categorized as SIMD or MIMD. In SIMD machines (single instruction stream, multiple data stream), processors are synchronized and execute a single sequence of instructions emanating from a single control unit, possibly on different data. Examples of SIMD machines are ICL DAP, Goodyear MPP, Connection Machine CM-2, and MasPar MP-1216. In MIMD machines (multiple instruction stream, multiple data stream), processors have independent control units and thus can execute different programs on different data.

The majority of existing parallel machines are MIMD; examples are the Sequent Symmetry, Encore MultiMax, Alliant FX/8, nCUBE, Intel iPSC/860, Intel Delta, and Connection Machine CM-5. Parallel architectures can also be classified according to their memory organization. In a shared memory architecture, memory is globally shared by all processors. Typically, this is accomplished by connecting the processors to the memory modules using a high-speed bus. Some examples are

Sequent Symmetry, Encore MultiMax, Alliant FX/8. The advantage of the shared memory architecture is that each processor has equal-time access to all shared memory locations. Therefore, data placement is not an important issue. However, this kind of architecture does not scale past a small number of processors (on the order of 50). In a distributed memory architecture, each processor is combined with a memory unit into a single node; nodes are connected using a scalable interconnection network (e.g., a ring, a mesh, or a hypercube) and communicate via message-passing. Some examples are Intel iPSC/860, nCUBE, Intel Delta, Connection Machine CM-5. The architecture is scalable to a very large number of processors (current machines contain thousands of processors). But a disadvantage of the architecture is the processor's non-uniform access to data (i.e., remote memory accesses take much longer than local memory accesses).

Distributed memory MIMD architectures are emerging as the consensus approach to scalable general-purpose parallel processing. A MIMD machine offers greater flexibility than a SIMD machine because it can execute different programs on different nodes, or it can execute different tasks of a single program on different nodes (the latter is sometimes referred to as SPMD or single program, multiple data). In addition, the distributed memory organization allows parallel machines to be built from off-the-shelf microprocessors and memory chips and to be scaled up to a large number of processors. However, programming distributed memory machines is more difficult than for shared memory machines. In this thesis we focus on distributed memory MIMD architectures.

2.2 The Issue of Grain Size Optimization

An important factor that determines program performance on a distributed memory parallel machine is the speed at which computation and communication can be

performed. Over the last decade, processor speeds have increased at the dramatic rate of 50% a year. On the other hand, communication speeds have not kept pace. To be sure, the bandwidth of interconnection networks has improved by employing better routing algorithms (e.g., wormhole routing), by using better packaging of parallel processors, or by simply increasing the number of wires in the links that make up the interconnection network. However, the cost of routing a message depends not only on its *transport time* (the time that it stays in the network) but also on the *overhead* spent in executing the operating system routines for sending and receiving the message. On contemporary machines, this software overhead is so large that it often dominates the transport time, even for messages traveling very long distances in the network. Typically, the message overhead is of the order of hundreds to a few thousands of processor clock cycles.

A parallel program can be viewed abstractly as a collection of tasks, where each task consists of a sequence of instructions and input and output parameters. A task starts execution only after all of its input parameters are available; output parameters are sent to other tasks only after the task completes execution. This notion of a task is called the "macro-dataflow model" by Sarkar [43] and is used by other researchers [42, 16, 27, 40, 49, 51]. In the macro-dataflow model, a parallel program is represented as a weighted directed acyclic graph (DAG) $G = (V, E, \mu, \lambda)$, where each $v \in V$ represents a task whose execution time is $\mu(u)$ and each directed edge (or arc) $(u, v) \in E$ representes time constraint that task u should complete its execution before task v can be started. In addition, u communicates data to v upon its completion; the delay incurred by this data transfer is $\lambda(u, v)$ if u and v reside in different processors and zero otherwise. In other words, task v cannot begin execution until all of its predecessor tasks have completed and it has received all data from these tasks.

The granularity of a task graph is an important parameter which we take into account when analyzing the performance of our algorithms. Basically there are two distinct strategies for scheduling: parallelizing tasks or sequentializing tasks. The trade-off point between parallelization and sequentialization is closely related to the granularity value: the ratio between the task execution time and communication time. If communication cost is too high, parallelization is not encouraged.

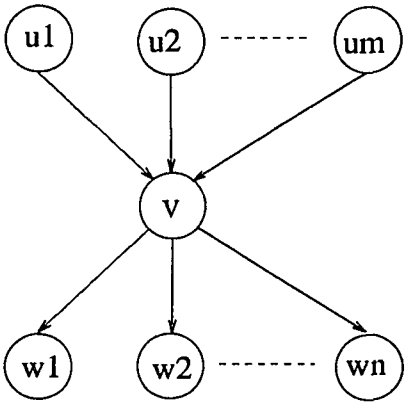


Figure 2.2 Definition of the task graph granularity.

We adopt the definition of granularity given in [18]. Let $G = (V, E, \mu, \lambda)$ be a weighted DAG. For a node $v \in V$ as shown in Figure 2.2, let

$$g_1(v) = \min\{\mu(u) | (u,v) \in E\} / \max\{\lambda(u,v) | (u,v) \in E\} \text{ and}$$

$$g_2(v) = \min\{\mu(w) | (v,w) \in E\} / \max\{\lambda(v,w) | (v,w) \in E\}.$$

The *grain-size* of v is defined as $\min\{g_1(v), g_2(v)\}$. The *granularity* of DAG G is given by $g(G) = \min\{g(v) | v \in V\}$. One can verify that for the DAG G of Figure 2.4, $g(G) = \frac{1}{7}$.

The high communication overhead in existing distributed memory parallel machines imposes a minimum threshold on program granularity below which performance degrades significantly. To avoid performance degradation, one solution would be to coalesce several fine grain tasks into single coarser grain tasks. This

reduces the communication overhead but increases the execution time of the (now coarser grain) tasks. Because of this inherent tradeoff, the goal is to determine the program granularity that results in the fastest total parallel execution time. This problem is called *grain size optimization.*

2.3 Grain Size Optimization Via Task Clustering

Grain size optimization can be viewed as the problem of scheduling the tasks of the program on the processors of the parallel machine such that the finish time of the last task (or "makespan of the schedule") is minimized. Much of the early work in scheduling algorithms considered only the task execution times and assumed zero communication times between interacting tasks. The survey papers by Coffman [9], Graham, et al. [20], and Lawler et al. [29] give excellent summaries of work in this area.

More recent work in scheduling algorithms explicitly consider inter-task communication times. The basic idea behind most of these algorithms is "task clustering", i.e., scheduling several communicating tasks in the same processor so that the communications between these tasks are realized as local memory accesses within the processor, instead of message transmissions across the interconnection network. In other words, the communication time between two tasks becomes zero when these tasks are mapped to the same processor. The result is a reduction in the message overhead, and hence total parallel execution time.

Researchers have investigated two types of task clustering algorithms, depending on whether or not task duplication (or recomputation) is allowed. In task clustering without duplication, the tasks are partitioned into disjoint sets or clusters and exactly one copy of each task is scheduled. In task clustering with duplication, a task may have several copies belonging to different clusters, each of which are

independently scheduled. In general, for the same DAG, task clustering with duplication produces a schedule with a smaller makespan (i.e., total execution time) than when task duplication is not allowed.

For example, for the *fork* DAG shown in Figure 2.3(a), the optimal makespan without task duplication is 16 while that with task duplication is 11. Note that when two communicating tasks are mapped to the same processor, the communication delay becomes zero because the data transfer is effectively a local memory write followed by a local memory read.

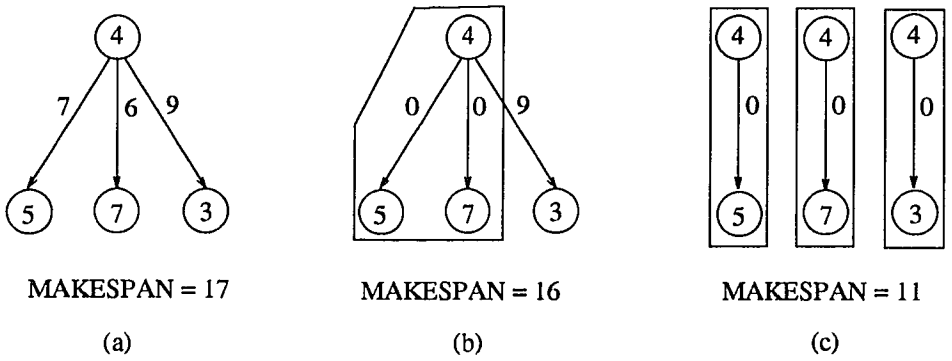


Figure 2.3 (a) A fork DAG; (b) optimal clustering without task duplication; (c) optimal clustering with task duplication.

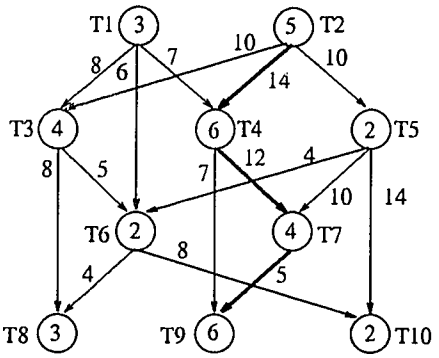


Figure 2.4 An example of DAG.

Figure 2.4 gives an example of DAG; the node weights denote the task execution times and the arc weights denote the communication delays. Thus, assuming that each task resides in a separate processor, the earliest time that task T4 can be started

is 19, which is the time it needs to wait until the data from task T2 arrives (the data from task T1 arrives earlier, at time 10). The makespan of the schedule is the length of the critical path, i.e., the path with the maximum sum of node and arc weights. In Figure 2.4, the critical path is indicated by the bold arcs; its length, and hence the makespan of the schedule, is 52.

A *clustering* of G is the mapping of the nodes in V onto *clusters*, where each cluster is a subset of V. If the clusters form a partition of V (i.e., they are pairwise disjoint) then the clustering is said to be *without duplication*. Similarly, if a node is mapped to more than one cluster (i.e., it has more than one *copy*) then the clustering is said to be *with duplication*. For example, for the DAG of Figure 2.4, the clustering $\Phi_1 = \{\{T_1, T_4, T_7, T_9\}, \{T_2, T_3, T_5, T_6, T_8, T_{10}\}\}$ is without duplication, while the clustering $\Phi_2 = \{\{T_1\}, \{T_2, T_4, T_5, T_7, T_9\}, \{T_2, T_3, T_5, T_6, T_8, T_{10}\}\}$ is with duplication; in the latter, nodes T_2 and T_5 each have two copies.

A *schedule* for a clustering Φ maps the clusters of Φ to processors and assigns to each node v a start time $s(v, p)$ on every processor p to which v is mapped. The schedule should satisfy the following condition for every node v: if v is mapped to processor p then, for every immediate predecessor u of v, there is some processor q to which u is mapped such that $s(v, p) \geq s(u, q) + \mu(u) + \lambda'(u, v)$, where $\lambda' = \lambda(u, v)$ if $p \neq q$. Let $s(v)$ =min$\{s(v, p) | v$ is mapped to processor p$\}$. The *makespan* of the schedule is given by max$\{s(v) + \mu(v) | v$ is a sink node$\}$.

A schedule S is optimal for a clustering Φ if for every other schedule S' for Φ, it is the case that $makespan(S) \leq makespan(S')$. We define the makespan of a clustering Φ as the makespan of its optimal schedule S. A clustering Φ is optimal for a DAG G if for every other clustering Φ' for G, $makespan(\Phi) \leq makespan(\Phi')$. Figure 2.5 gives a schedule for the clustering $\Phi_2 = \{\{T_1\}, \{T_2, T_4, T_5, T_7, T_9\}, \{T_2, T_3, T_5, T_6, T_8, T_{10}\}\}$ defined earlier. In the schedule, the three clusters are mapped to distinct processors; the value s_i beside the

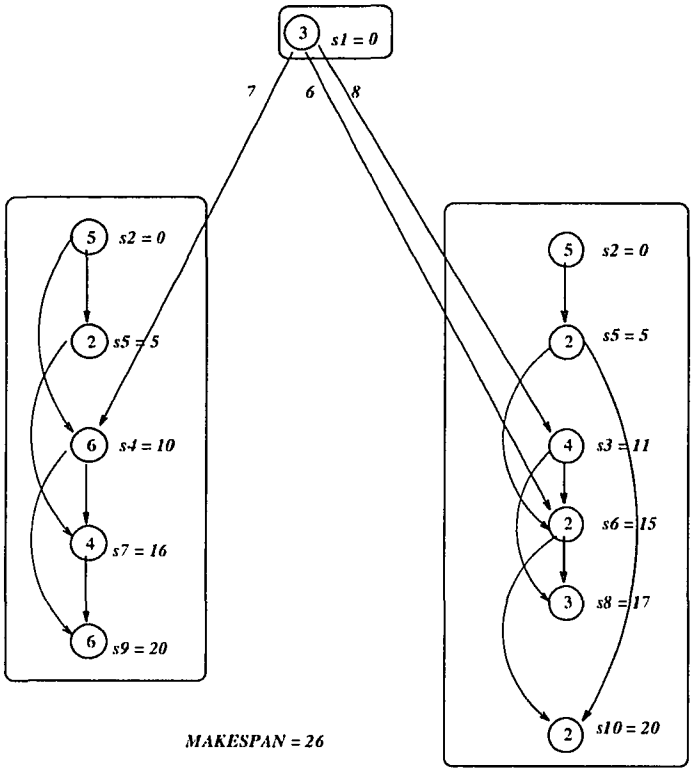


Figure 2.5 An optimal clustering with duplication for the DAG of Figure 2.4.

node denotes the start time of task T_i on the designated processor. The makespan of this schedule is 26. It turns out that this schedule is optimal for clustering Φ_2. It also turns out that Φ_2 is an optimal clustering for the DAG of Figure 2.4. Therefore, the shortest possible execution time for the DAG is 26.

Similar to the Figure 2.5 which gives an optimal schedule for clustering with task duplication, Figure 2.6 shows an optimal schedule for clustering without duplication for the example DAG in Figure 2.4. The clustering $\Phi_3 = \{\{T_1\}, \{T_2, T_4, T_5, T_7, T_9\}, \{T_3, T_6, T_8, T_{10}\}\}$ are mapped to three different processors. The makespan of this schedule is 26.

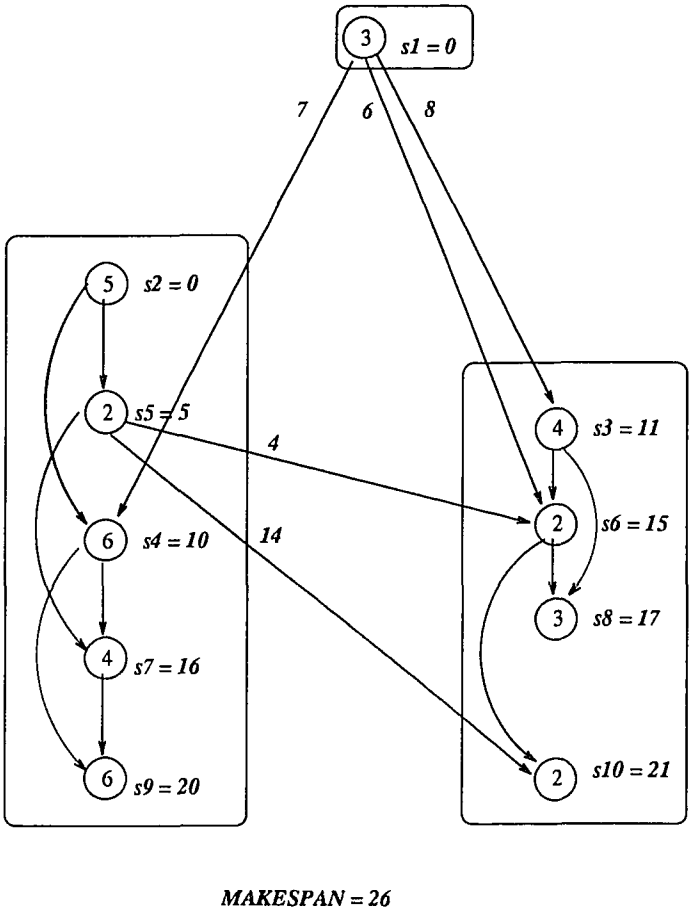


Figure 2.6 An optimal clustering without duplication for the DAG of Figure 2.4.

2.4 Previous Work on Task Clustering and Scheduling

In this section, we discuss existing heuristic algorithms for the task clustering and scheduling problem. Task clustering - with or without task duplication - is an NP-hard problem [7, 40, 43]. Consequently, practical solutions will have to sacrifice optimality for the sake of efficiency. Nonetheless, task clustering heuristic algorithms have a number of properties in common when they try to achieve the goal of finding an optimal clustering for a DAG G. They all perform a sequence of *clustering refinements* starting with an initial clustering (initially each task is assumed to be in a cluster). Each step performs a refinement of the previous clustering so that the final clustering satisfies or "near" to the original goals. The algorithms are non-

backtracking, i.e., once the clusters are merged in a refinement step, they cannot be unmerged afterwards.

A typical refinement step is to merge two clusters and *zero* the edge that connect them. Zeroing the communication cost on the edge between two clusters is necessary for reducing the makespan (or parallel time) of the schedule. The *makespan* is determined by the longest path in the scheduled graph. In other words, the makespan of a given schedule is equal to the time of the last task has been completely executed.

There are two important parameters in performing the refinement steps: the critical path of a task graph G and the earliest start time of each node $v \in V$. The *critical path* (CP) is the longest path in the task graph. In [18], Gerasoulis and Yang use *dominant sequence* (DS) instead of CP to represent the longest path of the scheduled task graph or the path whose length equals the actual makespan of the schedule. Nonetheless, the CP is so important that the heuristic algorithms rely on it for a global information of the task graph to guarantee the reduction of makespan in each refinement steps. We will show later the necessity of critical path as a global information. On the other hand, the earliest start time of a node $v \in V$ is the earliest time that node v can start execution for any clustering. If the execution of every node in G is started at its earliest start time, then the schedule must be optimal.

Assuming there is a list of available nodes (or tasks) ready for clustering at certain refinement step, heuristic algorithms have to make decisions on: (1) which node to take from the list; (2) where to put it. Usually the decisions are made according to cost functions and objectives of clustering heuristics. If the objective of clustering heuristic is to execute every node as early as possible, the cost function will then be the earliest start time and the strategy of allocation is to put the node into a cluster where the node can start execution as early as possible. However, if the objective is to minimize the makespan, then the critical path will be adopted as

the cost function and the node is put into a cluster that causes the minimum critical path.

We distinguish between two classes of task scheduling algorithms. The two-phase method performs a clustering first, under the assumption that there is an unbounded number of fully connected processors. When two tasks are assigned to the same cluster, they are executed in the same processor. At the second phase, clusters are merged and scheduled on the p physical processors if the number of clusters m is larger than p. Examples of clustering algorithms using the two-phase method are Sarkar [43], the DSC algorithm of Gerasoulis and Yang [18], Papadimitriou and Yannakakis [40], and Chung and Ranka [8]. On the other hand, the one-phase method schedules a task graph directly on the p physical processors. Scheduling algorithms in this class include the MCP algorithm of Wu and Gajski [50], and Kwok and Ahmad [28]. Kim and Browne [24, 25] have experimented with the two-phase method and the one-phase method. They found that the two-phase method results in significantly better schedules than the one-phase method.

2.4.1 Clustering with Task Duplication

For task clustering with duplication, Papadimitriou and Yannakakis [40] have developed a polynomial-time algorithm for arbitrary DAGs that generates a schedule whose makespan is at most twice optimal. Other algorithms that allow duplication, such as those given in Kruatrachue and Lewis [26], Chung and Ranka [8], and Kwok and Ahmad [28], do not give theoretical guarantees on performance as does the PY algorithm. Moreover, in terms of speed, the PY algorithm is also the fastest.

PY's algorithm

We consider the PY algorithm as a clustering heuristic based on earliest start time criterion. They use e values in their algorithm as the cost function. The basic idea of the PY algorithm is to minimize the start time of each node without considering

load balance. The way it constructs the clustering is that each node $v \in V$ is put into cluster C along with the nodes in the subset of v's ancestors that actually determine the $e(v)$. The algorithm can be described as follows:

1. For each node $v \in V$ consider the set of node v's ancestors, and for each such ancestors u compute the cost function $f(u) = e(u) + \mu(u) + \lambda(u, v)$.

2. Sort the ancestors in decreasing order: $f : f(u_1) \geq f(u_2) \geq \cdots \geq f(u_p)$.

3. Consider an integer j and suppose that $f(u_k) > j \geq f(u_{k+1})$.

   - Let $N_j(v)$ be the subdag of G consisting of all nodes u_i, $i \leq k$.
   - Find $L_j = \max_{i=1}^{k}[e(v_i) + \sum_{q=1}^{i} \mu(q)]$ such that $j \geq L_j$. Then $e(v) = j$.

4. The nodes of $N_{e(v)}(v)$ are executed by the processor of v.

In [40], page 326, the authors have shown that the makespan of the schedule produced by the PY algorithm is at most twice optimal. In terms of the quality of solutions generated, the PY algorithm is theoretically the best known polynomial-time algorithm for task clustering with duplication. However, its time complexity is quite high: $O(|V|^2(|V|lg|V| + |E|))$ time for a DAG with $|V|$ nodes (or tasks) and $|E|$ arcs. The main source of complexity in the algorithm is the method used to find, for each node v, the cluster that allows v to be executed as early as possible. To find this cluster, the algorithm keeps track of as many as $|V|$ candidate clusters, each of which takes $O(|V|lg|V| + |E|)$ time to process.

2.4.2 Clustering without Task Duplication

For task clustering without duplication, several polynomial-time algorithms have been proposed. Broadly speaking, these algorithms are based on three different heuristics: (1) critical path analysis [18, 24, 43, 50]; (2) priority-based list scheduling [2, 22, 30, 42]; and (3) graph decomposition [33]. Recently in [34], the empirical

performance of these algorithms were compared based on ten DAGs that model the structure of several practical application. Nonetheless, none of the algorithms have provable guarantees an upper bound on the quality of schedules they generate, relative to an optimal schedule. Some algorithms are guaranteed to work well for special DAGs. For coarse grain DAGs (i.e., DAGs whose task execution times are larger than the inter-task communication times), the DSC (Dominant Sequence Clustering) algorithm of [18] has been shown by the authors to give a schedule whose makespan is at most twice optimal, under the assumption that the number of processors is unbounded. The ETF (Earliest Task First) algorithm of [22] gives a schedule whose makespan is at most $(2 - 1/n)M_{opt} + C$, where M_{opt} is the optimal makespan on n processors without considering communication delays and C is the maximum communication delay along any chain of nodes in the DAG.

DSC algorithm

Gerasoulis and Yang [18] presented a clustering algorithm based on the reduction of the critical path. At each stage in their clustering algorithm, they define that the set of nodes which are in the longest path of scheduled task graph at that stage is called the *Dominant Sequence* (DS). The dominant sequence, however, reduces to the critical path for linear clustering.

The basic idea of the DSC is to identify the DS at each step and then zero edges in that DS. It is based on the following observations:

- Zeroing one edge in the CP will change CP in the next stage.
- Reducing DS at each step locally will let DS be computed incrementally without having to traverse the entire graph again.

Let $tlevel(n)$ be the length of the longest path from a source node to n in the scheduled DAG (excluding $\mu(n)$) and $blevel(n)$ be the length of the longest path from n to a sink node (including $\mu(n)$). DSC can then be described as follows:

1. Compute cost function *blevel* for each node and set $tlevel = 0$ for each *free* node.

2. For each DS in an examined task graph G^*:

   - Apply edge zeroing operation on DS that result in the largest possible decrease of makespan.
   - If no zeroing is accepted, node remains in a unit cluster. Apply edge zeroing recursively to the subDS (next longest path).

3. Update the DS, repeat step 2 until all edges are examined.

By localizing the zeroing, the algorithm computes DS incrementally and eventually has a complexity of $O((|E| + |V|)lg|V|)$ which is faster than any other clustering algorithms in this problem. In [51, 16, 17], the authors have compared their DSC algorithm with other algorithms for task clustering without duplication and shown that the DSC algorithm outperforms the other algorithms in terms of speed and (empirical) solution quality.

2.4.3 Dynamic Scheduling

All the heuristics mentioned previously are modeled as static task graphs. On the other hand, scheduling algorithms for dynamic task graphs have received little attention although programs for an increasing number of scientific application naturally fall into this class. There are some basic approaches to dynamic scheduling: (1) Unconstrained FIFO; (2) Balance-constrained; (3) Cost-constrained; (4) Hybrids scheduling.

The most elementary approach to dynamic scheduling assumes no a priori knowledge of the parallel program. In Unconstrained FIFO scheduling $p + 1$

processors are used: one PE runs a schedule that dispatches tasks on a first-in-first-out (FIFO) basis to all other p PEs. The schedule produced by this heuristic is often far from optimal.

A balanced-constrained heuristic attempts to rebalance the loads on all processors by periodically shifting waiting tasks from one waiting queue to another. In distributed memory architectures, this could involve many realignments. It is not guaranteed to always find the minimum time. A cost-constrained heuristic works as follows. It performs the balance-constrained heuristic locally to identify candidate tasks to be moved. These tasks are then checked for communication costs to see if it is greater than the decrease in execution time before they are moved.

A hybrid scheduling which is a combination of static and dynamic schedulers can be used in the case of loop-back and branch. In this type of problem, we might know the probabilities of a branch or loop estimated by profiling the program on a number of actual runs. Then we can perform a static scheduling on these graph and encode them into a dynamic scheduler table, and apply the appropriate static schedule at run time.

CHAPTER 3

CLUSTERING STATIC TASK GRAPHS WITH DUPLICATION

For task clustering with duplication, the Papadimitriou and Yannakakis' algorithm is theoretically the best known polynomial-time algorithm in terms of the quality of solutions generated. It has been shown by the authors that their algorithm generates a schedule whose makespan is at most twice optimal. In this chapter, we present a better algorithm for this problem which, for a task graph with arbitrary granularity g, produces a schedule whose makespan is at most $(1 + 1/(1 + g))$ times optimal. Therefore, the quality of the schedule improves as the granularity of the task graph becomes larger. For example, if the granularity is at least $1/2$, the makespan of the schedule is at most $5/3$ times optimal. For a task graph with $|V|$ tasks and $|E|$ inter-task communication constraints, the algorithm runs in $O(|V|(|V|lg|V| + |E|))$ time, which is $|V|$ times faster than the PY algorithm for this problem. Similar algorithms are developed that produce: (1) optimal schedules for coarse grain graphs; (2) 2-optimal schedules for trees with *no* task duplication; and (3) optimal schedules for coarse grain trees with *no* task duplication.

3.1 CASS-I Algorithm

This section presents a greedy algorithm that finds a clustering for a given DAG $G = (V, E, \mu, \lambda)$. We prove that the clustering $\Phi(G)$ produced by the algorithm is "good" in the following sense: If $g(G) \geq (1 - \varepsilon)/\varepsilon$ for some $0 < \varepsilon \leq 1$, then the makespan of $\Phi(G)$ is at most $(1 + \varepsilon)$ times the makespan of the optimal clustering for G. As a corollary, for a DAG with *arbitrary* granularity (i.e., $g(G) \geq 0$), the clustering produced by the algorithm has a makespan which is at most twice optimal, thus matching the bound of the PY algorithm [40]. However, as $g(G)$ increases, the

bound gets better. For example, if $g(G) \geq \frac{1}{2}$, the makespan of the clustering is at most $\frac{5}{3}$ times optimal.

The basic idea behind the algorithm is similar to the PY algorithm. For each $v \in V$, we first compute a lower bound $e(v)$ on the earliest possible start time of v. This is accomplished by finding a cluster $C(v)$ containing v that allows v to be started as early as possible when all the nodes in $C(v)$ are executed on the same processor and all other nodes in $V - C(v)$ are executed on other processors. This cluster can be determined using a simple greedy algorithm which (unlike the PY algorithm) grows the cluster one node at a time. Once the clusters are determined, they are mapped to processors in a simple way, and we show that this mapping has a schedule whose makespan is "good" in the sense described in the previous paragraph.

3.1.1 Computing the e Value

The e values are computed in topological order of the nodes of G. For a source node, its e value is equal to zero. For any other nodes, its e value is computed after all of its ancestors have been assigned e values. Consider a node v all of whose ancestors have been assigned e values, and suppose we wish to compute $e(v)$. Since $e(v)$ is a lower bound on the start time of v, it suffices to look at clusters C consisting of v and a subset of its ancestors. If a cluster C' contains a node w which is not an ancestor of v, removing w from C' results in a cluster in which v can be started possibly sooner, but never later, than v's start time in C'.

Let C be a cluster consisting of node v and a subset of its ancestors $\{u_1, \ldots, u_k\}$. We wish to find a lower bound $e_C(v)$ on the earliest start time of v assuming that all nodes in C are executed on the same processor. Ignore for the moment the arcs that cross C, i.e., those that connect nodes outside of C to nodes inside of C. What is the earliest time that v can be scheduled? Clearly, the answer is the makespan of the optimal schedule for the one-processor scheduling problem with release times for the

instance $\{u_1, \ldots, u_k\}$ with $e(u_i)$ and $\mu(u_i)$ being the release time and execution time, respectively, of task u_i. This problem is solved optimally by the greedy algorithm that executes the tasks in nondecreasing order of release times. Therefore, for the cluster C,

$$e_C(v) \geq GREEDY - SCHEDULE(C - \{v\}) \tag{3.1}$$

where GREEDY-SCHEDULE($\bullet$) returns the makespan of the one-processor schedule for the set of tasks specified by its argument.

Next consider the set of arcs that cross C. For such an arc (u, w), define its c *value* as $c(u, w) = e(u) + \mu(u) + \lambda(u, w)$. Node v cannot be scheduled before time $c(u, w)$ because there is a path from u to v through node w. Therefore,

$$e_C(v) \geq MAX - C - VALUE(C) \tag{3.2}$$

where MAX-C-VALUE(C) is the maximum c value among the arcs that cross C.

From (3.1) and (3.2), it follows that for a given cluster C,

$$e_C(v) \geq \max \left\{ \begin{array}{c} GREEDY - SCHEDULE(C - \{v\}) \\ MAX - C - VALUE(C) \end{array} \right\}$$

and

$$e(v) \geq \min_C \{e_C(v)\} \tag{3.3}$$

The problem is to find the cluster C for which $e_C(v)$ is minimum. We show that

this cluster can be found using a simple greedy algorithm. Starting with node v, the algorithm "grows" the cluster a node at a time and checks if the new cluster can potentially decrease the current estimate for the e value of v. If growing the cluster can only increase the e value, the algorithm stops and returns the minimum e value obtained.

Suppose we have found a candidate cluster C; hence $e_C(v)$ satisfies Equation (3.3). We have the following two cases:

Case 1: MAX-C-VALUE(C) > GREEDY-SCHEDULE($C - \{v\}$). Let (u, w) be an arc that crosses C such that $c(u, w)$ = MAX-C-VALUE(C). Since $e_C(v) \geq c(u, w)$, an e value less than $e_C(v)$ cannot be obtained as long as node u is outside of the cluster C. Therefore, C must be grown to include node u.

Case 2: GREEDY-SCHEDULE($C - \{v\}$) $\geq$ MAX-C-VALUE(C). Since $e_C(v) \geq$ GREEDY-SCHEDULE($C - \{v\}$), then adding *any* new node x to C cannot decrease the e value because GREEDY-SCHEDULE($C \cup \{x\} - \{v\}$) $\geq$ GREEDY-SCHEDULE($C - \{v\}$).

Case 1 gives the criterion for growing the candidate cluster while case 2 gives the stopping criterion. The complete algorithm for computing $e(v)$ is given as Algorithm COMPUTE-E-VALUE below. For the DAG of Figure 2.4, Figure 3.1 shows the e values computed by the algorithm. Figure 3.2 illustrates how the e value of node 10 is computed.

1. **Algorithm** COMPUTE-E-VALUE(v, G)
2. **begin**
3. **if** v is a source node **then return** (0);
4. $C \leftarrow \{v\}$;
5. $m \leftarrow 0$;

6. $c \leftarrow$ MAX-C-VALUE(C);

7. $e \leftarrow c$;

8. **while** $m < c$ **do**

9. **let**(u, w) be an arc such that $u \notin C, w \in C$, and $c = c(u, w)$;

10. $C \leftarrow C \cup \{u\}$;

11. $m \leftarrow$ GREEDY-SCHEDULE($C - \{v\}$);

12. $c \leftarrow$ MAX-C-VALUE(C);

13. $e \leftarrow$min{e,max{m, c}};

14. **endwhile;**

15. **return** (e);

16. **end** COMPUTE-E-VALUE.

3.1.2 Constructing the Schedule

Algorithm COMPUTE-E-VALUE (v, G) can be easily modified so that it returns, in addition to $e(v)$, the corresponding cluster $C(v)$. We now describe how to construct a clustering $\Phi(G)$ for G whose makespan is at most $(1+\varepsilon)$-optimal if $g(G) \geq (1-\varepsilon)/\varepsilon$.

$\Phi(G)$ is constructed by visiting the nodes of G in reverse topological order (i.e., from sink nodes to source nodes). Initially, $\Phi(G) = \emptyset$ and the sink nodes of G are "marked". The following steps are then performed until there are no more marked nodes:

1. Pick a marked node v and add $C(v)$ to $\Phi(G)$.

2. Unmark v and mark all nodes u for which there is an arc (u, w) such that $u \notin C(v)$ and $w \in C(v)$.

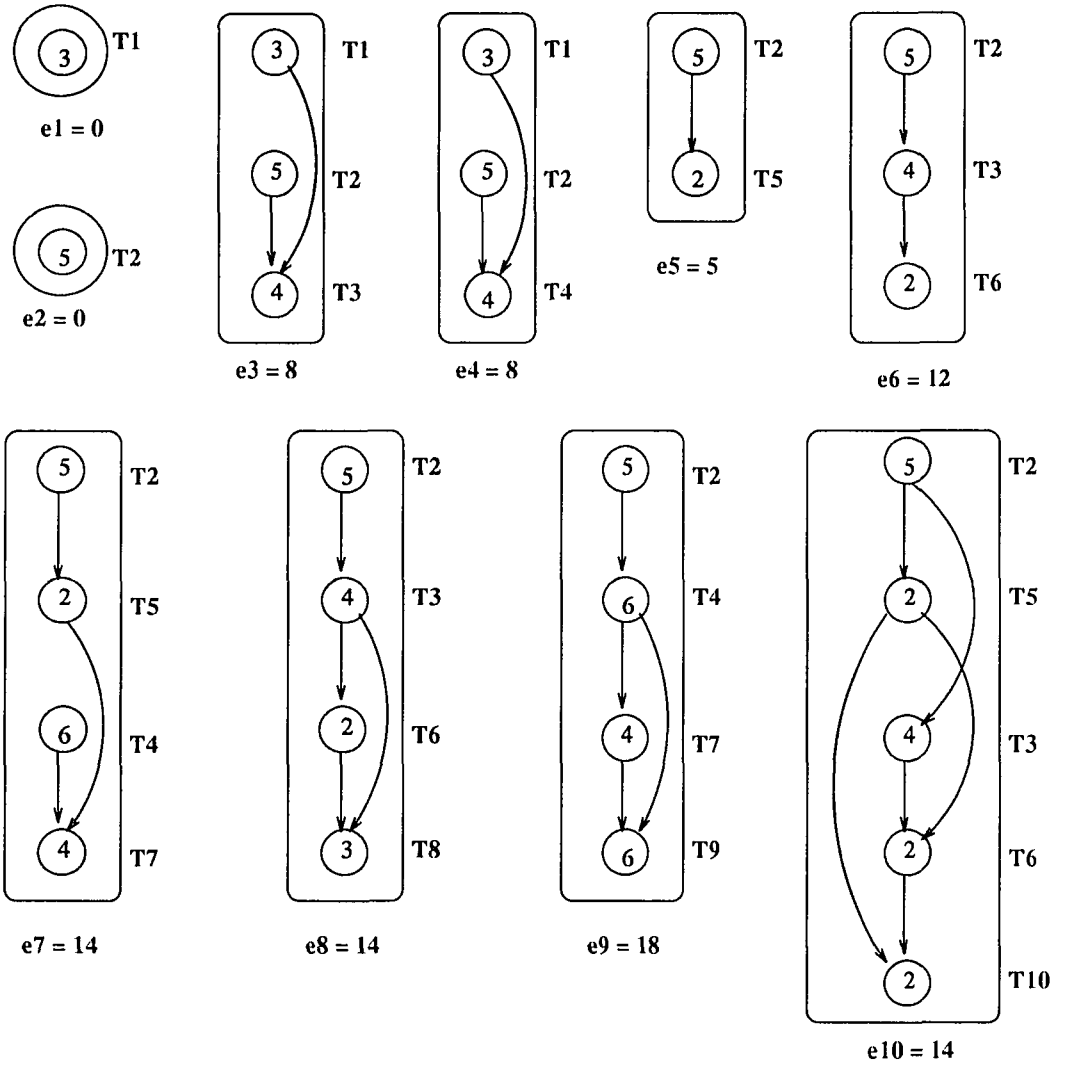


Figure 3.1 The e values and clusters for the DAG of Figure 2.4.

To schedule $\Phi(G)$, we map each cluster in $\Phi(G)$ to a distinct processor and execute the nodes mapped to the same processor in nondecreasing order of their e values. If node w is mapped to processor p, we let $s(w,p)$ denote the start time of w in p. Note that each processor p holds exactly one cluster and that this cluster is $C(v)$ for some marked node v. Moreover, v is the last node executed in this cluster.

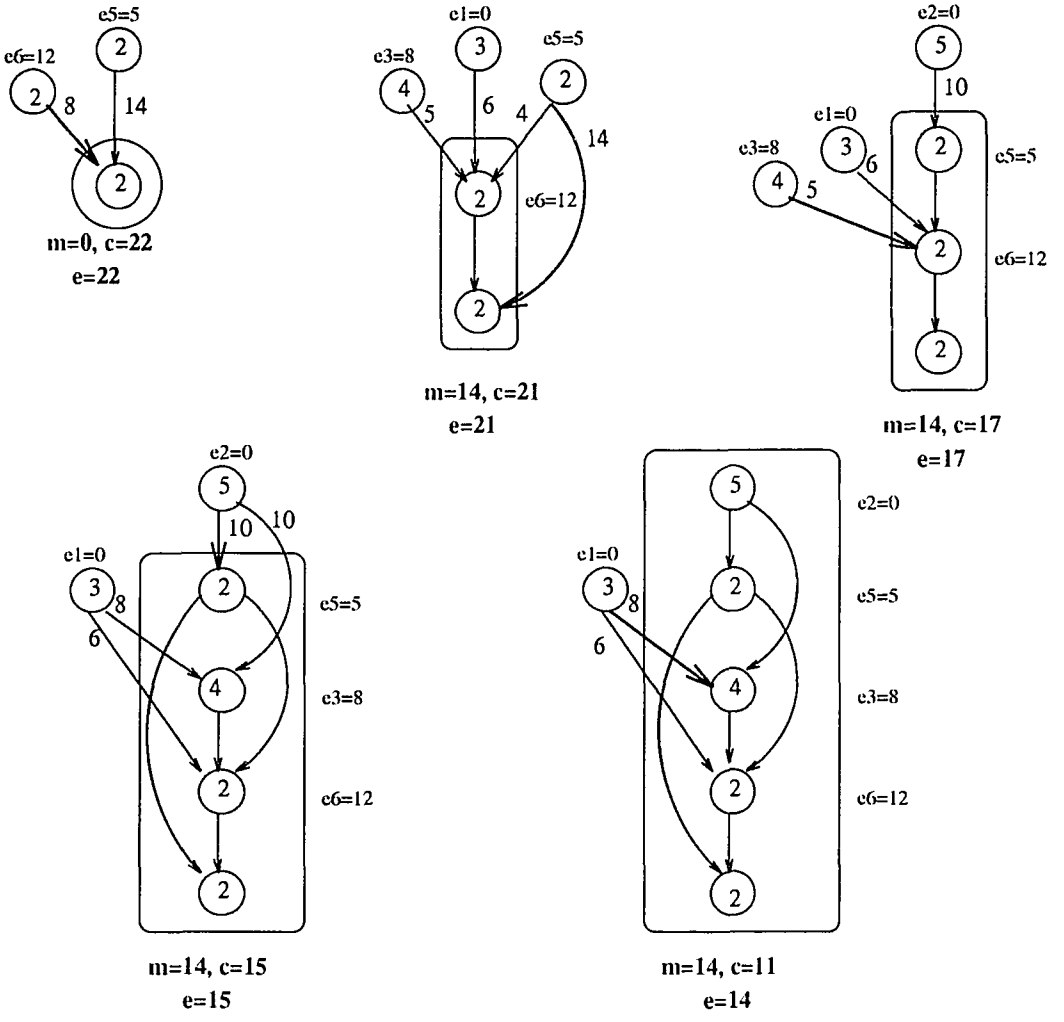


Figure 3.2 Computing the e value of node 10; critical arcs are in bold.

3.2 Performance Bounds for CASS-I

Let v be a non-source node. In the algorithm, each iteration of the **while** loop chooses an arc in step 9 for inclusion in the candidate cluster C. Call such arcs *critical*. Clearly, the set of critical arcs forms a tree T with root v, as illustrated in Figure 3.3. It follows that, for any cluster of nodes that includes v but excludes some other nodes in T, there is at least one critical arc that crosses it. Let m_i and c_i be the m and c values, respectively, that are computed in lines 11 and 12 at iteration i

of the **while** loop, and let m_0 and c_0 be the initial values before entering the loop. Furthermore, let $c_i = \max\{m_i, c_i\}$. Finally, let t be the last iteration of the loop.

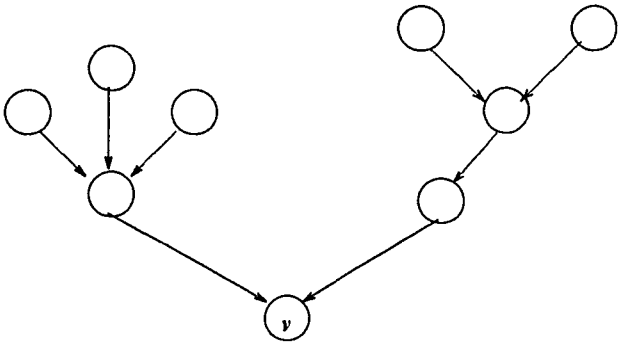


Figure 3.3 The set of critical arcs forms a tree T with root v.

The following fact is obvious.

Fact 1. For $0 \leq i < t$, $m_i < c_i$; and $m_t \geq c_t$.

Let k be the least integer such that $c_k = \min_{0 \leq i \leq t}\{c_i\} = e(v)$. From Fact 1, it follows that for $0 \leq i < t, c_i = \max\{m_i, c_i\} = c_i \geq e_k$. Similarly, $m_t = max\{m_t, c_t\} = c_t \geq e_k$. Therefore, we have

Fact 2. For $0 \leq i < t, c_i \geq e(v)$; and $m_t \geq e(v)$.

We are now ready to prove the following.

Theorem 1 *Algorithm COMPUTE-E-VALUE(v, G) returns a lower bound e(v) on the earliest start time of node v.*

Proof. The proof is by induction on the depth of node v. The theorem is obviously true for source nodes. So let v be a non-source node and assume that the theorem holds for all of v's ancestors. Suppose that the algorithm returns $e(v) = e_k$, the e value computed at iteration k of the **while** loop. Let C_k be the cluster of nodes constructed at this iteration. Suppose to the contrary that there is another cluster C' containing v in which v can be executed at time $e' < e_k = e(v)$. Then all nodes in C_k must be in C'. If not, there is some critical arc that lies inside C_k but crosses

C'. The c value of this critical arc is c_i for some $i, 0 \leq i < k \leq t$. Therefore, $e' \geq c_i$ which by Fact 2 implies $e' \geq c_k = e(v)$, a contradiction. Therefore, $C_k \subseteq C'$. Next consider the following two cases:

Case 1 : $k = t$. Since C' includes all nodes in C_t then $e' \geq m_t$. But by fact 1, $m_t = \max\{m_t, C_t\} = e_i$; hence $e' \geq e_t = e(v)$, which is a contradiction.

Case 2 : $k < t$. Let (u, w) be an arc that crosses C_k whose c value equals c_k. Then C' must contain (u, w). If not, then (u, w) must cross C' (since $w \in C'$). But this implies that $e' \geq c_k = max\{m_k, c_k\} = e_k = e(v)$, which is a contradiction. Now suppose (u, w) is in C'. Then *all* nodes in C_t must be in C'. Otherwise, there will be a critical arc that crosses C' whose c value is c_j for some $k < j < t$. But then $e' \geq c_j \geq e_k = e(v)$ (by Fact 2), which is again a contradiction. Finally, if C' includes all nodes in C_t, we arrive at the contradiction that $e' \geq m_t \geq e_k = e_v$ (using Fact 2).

We now prove the following.

Lemma 1 *Let v be a marked node such that $C(v)$ is mapped to processor p. If $g(G) \geq (1 - \varepsilon)/\varepsilon$ for some $0 < \varepsilon \leq 1$, then $s(v, p) \leq (1 + \varepsilon)e(v)$.*

Proof. The proof is by induction on the depth of marked node v. The theorem is true for all marked nodes v that do not have ancestors which are also marked nodes because in this case $s(v, p) = e(v)$. Now consider a marked node v and suppose that the theorem holds for all of its ancestors that are also marked nodes. For each node $w \in C(v)$, let (u, w) be an arc with the maximum c value among all arcs that cross $C(v)$ and ends at w. Thus u is a marked node. It follows that w can be started at time

$$s(w, p) \leq (1 + \varepsilon)e(u) + \mu(u) + \lambda(u, w).$$

since u is a predecessor of w, $e(w) \geq e(u) + \mu(u)$. Therefore,

$$s(w, p) \leq e(w) + \varepsilon e(u) + \lambda(u, w).$$

Now $\mu(u)/\lambda(u,w) \geq (1-\varepsilon)/\varepsilon$; thus, $\varepsilon[\mu(u)+\lambda(u,w)] \geq \lambda(u,w)$. It follows that

$$\begin{aligned} s(w,p) &\leq e(w) + \varepsilon[e(u)+\mu(u)+\lambda(u,w)]. \\ &\leq e(w) + \varepsilon e(v), \end{aligned}$$

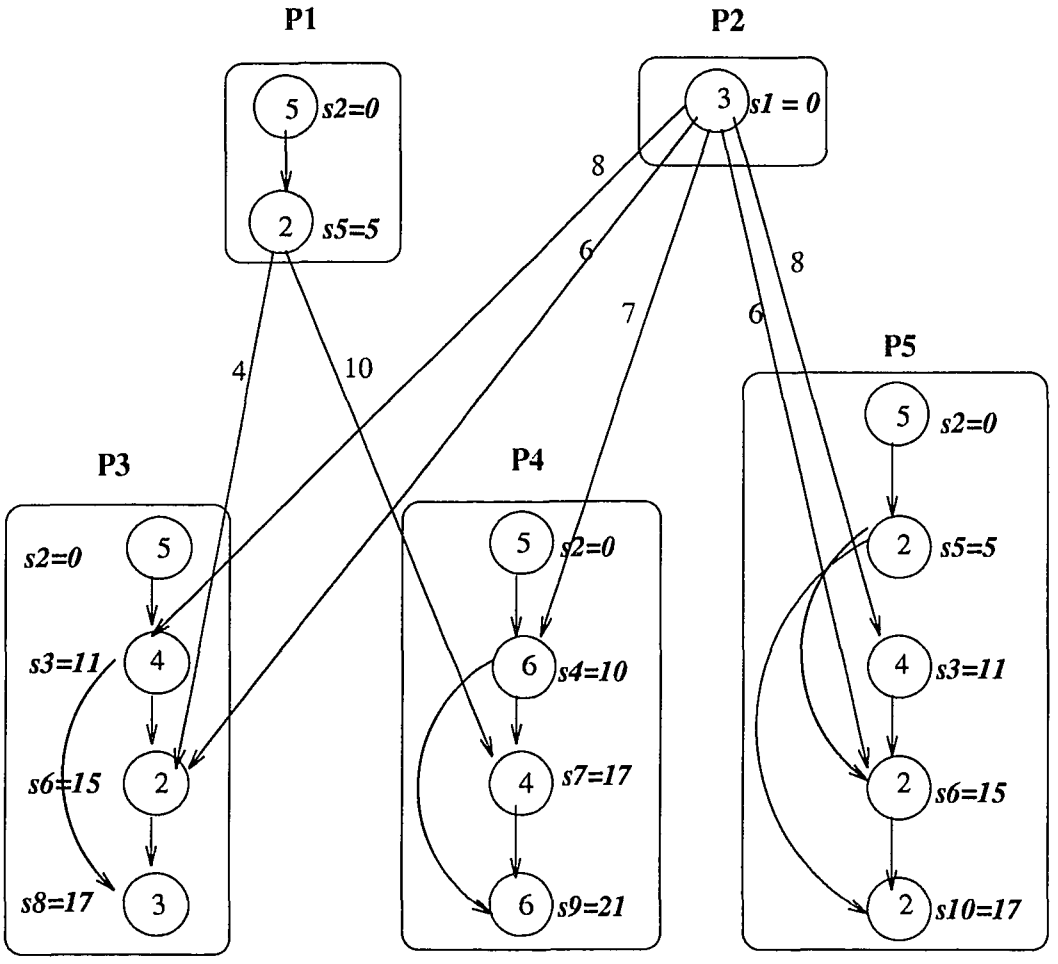
because $e(v) \geq e(u)+\mu(u)+\lambda(u,w)$. The last inequality implies that every node in $C(v)$ can be started at its e value plus a delay of at most $\varepsilon e(v)$. Therefore, v can be started at time $s(v,p) \leq$GREEDY-SCHEDULE$(C-\{v\}) + \varepsilon e(v) \leq (1+\varepsilon)e(v)$.

Theorem 2 *If $g(G) \geq (1-\varepsilon)/\varepsilon$ for some $0 < \varepsilon \leq 1$, then the makespan of $\Phi(G)$ is at most $(1+\varepsilon)$ times the makespan of an optimal clustering for G.*

Proof. Let M_{opt} be the makespan of an optimal clustering for G. Then $M_{opt} \geq \max\{e(v)+\mu(v)\}$, over all sink nodes v of C. Since every sink node v is a marked node, then by Lemma 1,

$$\begin{aligned} makespan(\Phi(G)) &\leq \max\{(1+\varepsilon)e(v)+\mu(v)\} \\ &\leq \max\{(1+\varepsilon)[e(v)+\mu(v)]\} \\ &= (1+\varepsilon)\max\{e(v)+\mu(v)\} \\ &\leq (1+\varepsilon)M_{opt}. \end{aligned}$$

Figure 3.4 shows the clustering constructed by the procedure for the DAG G of Figure 2.4. The figure also shows the start times of the nodes when the clusters are mapped to distinct processors. The makespan of the clustering is 27. On the other hand, the makespan of an optimal clustering is at least $e(T_9)+\mu(T_9) = 18+6 = 24$. The granularity of G is $g(G) = \frac{1}{7}$. Setting $\frac{1}{7} = (1-\varepsilon)/\varepsilon$ gives $\varepsilon = \frac{7}{8}$. Thus, $27 \leq (1+\frac{7}{8}) * 24 = 45$ as predicted by Theorem 2. Note that while 24 is a lower bound on the optimal makespan, the optimal makespan is 26, as shown in Figure 2.5. Therefore, the clustering produced by the algorithm is actually closer to optimal than predicted.



MAKESPAN = 27

Figure 3.4 A clustering and schedule for the DAG of Figure 2.4.

3.3 Complexity Analysis of CASS-I

In this section, we describe the implementation details and derive the time complexity of the algorithm. The runtime of Algorithm COMPUTE-E-VALUE depends on how the functions MAX-C-VALUE and GREEDY-SCHEDULE are computed. MAX-C-VALUE(C) is computed as follows: we maintain a Fibonacci heap $H[10]$ whose elements are the nodes of the DAG. For each node u we associate a value $key[u]$ which equals the maximum c value among all arcs that cross cluster C and emanate from

u. (If no such arc exists, $key[u] = -\infty$). The following operations are performed on H.

- EXTRACT-MAX(H): deletes from H the node with largest key.
- INCREASE-KEY(H, x, k): increases the key of node x in H to the value k.

The algorithm grows the cluster C by adding a node u from which emanates an arc with the maximum c value that crosses C. This node u is obtained by performing EXTRACT-MAX(H). Now, adding u to C reveals new arcs (x, u) that cross C. Therefore, heap H is updated by performing INCREASE-KEY($H, x, c(x, u)$) for each such arc (x, u).

EXTRACT-MAX is executed at most $|V|$ times. INCREASE-KEY is called once for each new arc that crosses C and hence is executed at most $|E|$ times. For a Fibonacci heap with $|V|$ elements, an EXTRACT-MAX operation can be performed in $O(lg|V|)$ amortized time and an INCREASE-KEY operation in $O(1)$ amortized time. Therefore, excluding the calls to function GREEDY-SCHEDULE, the algorithm runs in $O(|V|lg|V| + |E|)$ time.

Next consider the implementation of function GREEDY-SCHEDULE. In the algorithm, each subsequent call to GREEDY-SCHEDULE adds a single node (task) to the argument set. Moreover, the makespan of the greedy schedule is obtained by executing the tasks in the set in nondecreasing order of e values. Therefore, if the tasks are initially sorted, a new task can be inserted in the sorted list using binary search. However, although insertion can be done in $O(lg|V|)$ time, computing the makespan of the schedule for the new list will take $O(|V|)$ time.

By using a 2-3 tree T [1], we can reduce the time to compute the makespan to $O(lg|V|)$. In T, every internal node has either two or three children and all leaves are at the same distance from the root. Given a set of tasks and their e values, we store the tasks in the leaves of T in sorted order, i.e., arranged from left to

right in nondecreasing order of e values. We say that node α "owns" the sublist of tasks stored in the subtree rooted at α. Node α contains the following pieces of information:

- $e[\alpha]$: The maximum e value among all tasks owned by α.
- $s[\alpha]$, $f[\alpha]$: If the tasks owned by α are scheduled greedily, then $s[\alpha]$ is the start time of the first task executed and $f[\alpha]$ is the finish time of the last task executed. (Note the $f[\alpha]$ is also the makespan of the greedy schedule for this sublist of tasks.)
- $d[\alpha]$: The idle time in the greedy schedule for the sublist tasks owned by α; i.e., the number of time units between $s[\alpha]$ and $f[\alpha]$ during which no task is being executed.

Figure 3.5 illustrates how the 2-3 tree T evolves for the given sequence of tasks. To insert a new task x, we traverse the tree downward from the root to locate the point of insertion, insert a new leaf corresponding to task x, then traverse the tree upward towards the root to update the information of the nodes affected by the insertion. Observe that an insertion may cause some nodes along the traversed path to have four children, in which case the node is split into two nodes, each with two children. The details of insertion and node splitting can be found in [1].

Figure 3.6 shows how a node's key values are updated, given the key values of its children. The proof of correctness is straightforward and is omitted here. Note that only the nodes along the root-to-leaf path are updated and that an update takes constant time.

Since T has at most $|V|$ leaves, its depth is $O(lg|V|)$ and hence insertion takes $O(lg|V|)$ time. Moreover, the makespan of the greedy schedule for the tasks currently stored in T can be found in $O(1)$ time as it is simply $f[root(T)]$. It follows that the calls to GREEDY-SCHEDULE in Algorithm COMPUTE-E-VALUE contribute

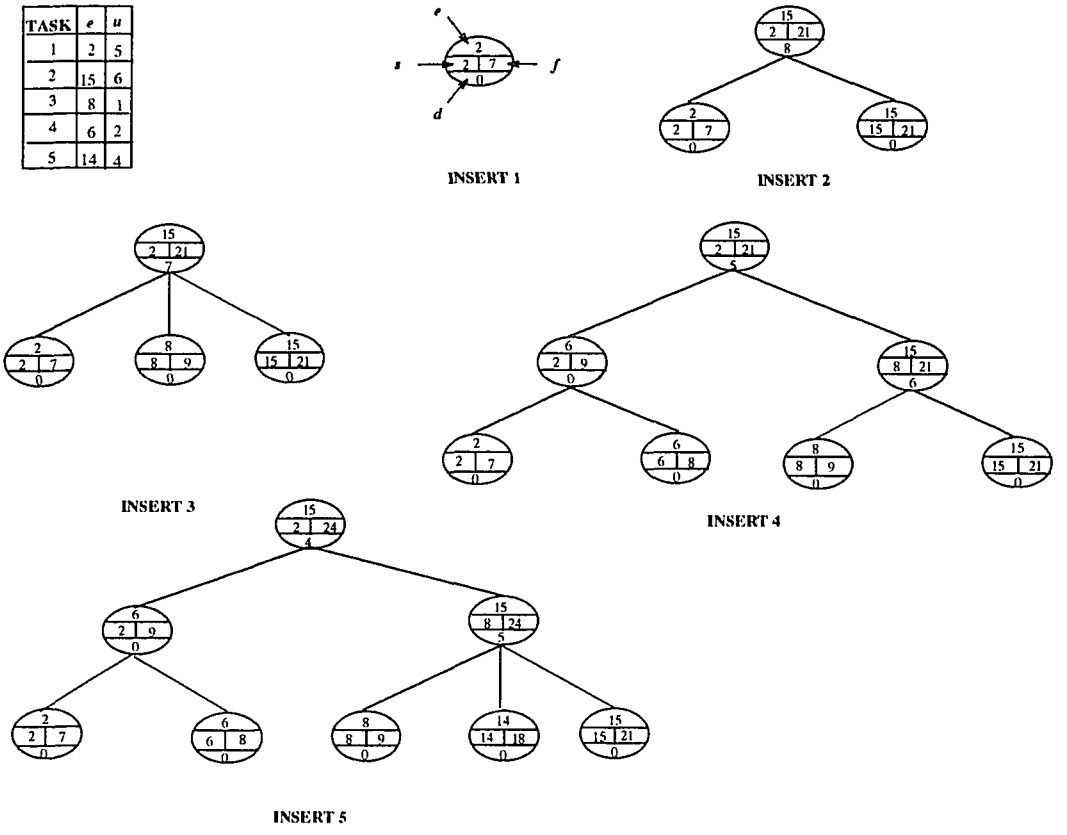


Figure 3.5 The 2-3 tree T.

$O(|V|lg|V|)$ to the total runtime. Therefore, Algorithm COMPUTE-E-VALUE runs in $O(|V|lg|V|+|E|)$. Since the algorithm is called $|V|$ times, computing the e values of all nodes takes $O(|V|(|V|lg|V|+|E|))$ time.

Finally, once the e values are computed, the clustering can be constructed in $O(|V|+|E|)$ time. Therefore, the entire task clustering algorithm takes $O(|V|(|V|lg|V|+|E|))$ time.

3.4 Special Cases

3.4.1 Coarse Grain DAGs

A DAG G is *coarse grain* if $g(G) \geq 1$; otherwise it is *fine grain*. For coarse grain DAGs, the task clustering algorithm of the previous section produces a clustering

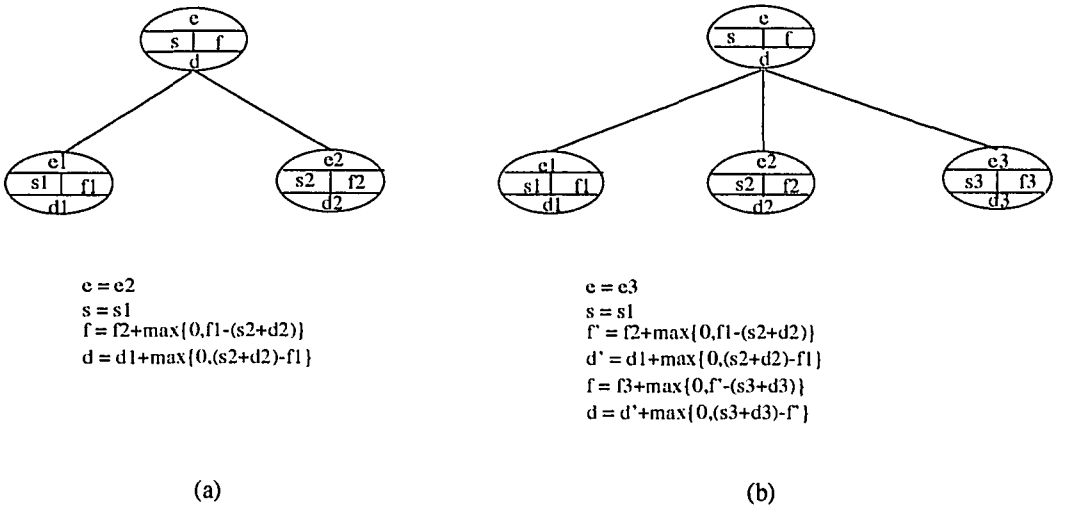


Figure 3.6 Updating a node with (a) two children;(b) three children.

whose makespan is at most 1.5 times optimal. We show that by slightly modifying the algorithm an optimal clustering can be obtained. Before showing this result, we prove some properties of coarse grain DAGs.

Let $G = (V, E, \mu, \lambda)$ be a coarse grain DAG. For $v \in V$, let $e(v)$ be the e value returned by Algorithm COMPUTE-E-VALUE(v, G) and let $C(v)$ be the corresponding cluster. For nodes $u, v, \in V$, call u a *critical predecessor* of v if and only if $c(u, v) = max\{c(w, v) | (w, v) \in E\}$.

Lemma 2 *If u is a critical predecessor of v, then $e(v) \geq max\{e(u)+\mu(u), max\{c(w, v)$ $|(w, v) \in E$ and $w \neq u\}\}$.*

Proof. The claim obviously holds if $e(u) + \mu(u) \geq max\{c(w, v)|(w, v) \in E$ and $w \neq v\}$ because $e(v) \geq e(u) + \mu(u)$ (u is a predecessor of v). So assume to the contrary that $e(v) < c(x, v)$ for some $x \neq u$. (Note that $c(u, v) \geq c(x, v)$.) Then x and u must both be executed on the same processor as v. Therefore, $e(v) \geq$ GREEDY-SCHEDULE(u, x). We have two cases:

Case 1 : $e(u) \geq e(x)$. Then

$$e(v) \geq max\{e(u) + \mu(u), e(x) + \mu(x) + \mu(u)\}$$

$$\geq e(x) + \mu(x) + \mu(u)$$
$$\geq e(x) + \mu(x) + \lambda(x,v), \text{ since } g(v) \geq 1$$
$$= c(x,v)$$
$$> e(v), \text{ a contradiction.}$$

Case 2 : $e(x) > e(u)$. Then

$$\begin{aligned} e(v) &\geq max\{e(x) + \mu(u), e(u) + \mu(u) + \mu(x)\} \\ &\geq e(u) + \mu(u) + \mu(x) \\ &\geq e(u) + \mu(u) + \lambda(u,v), \text{ since } g(v) \geq 1 \\ &= c(u,v) \\ &\geq c(x,v) \\ &> e(v), \text{ a contradiction.} \end{aligned}$$

A cluster of nodes $C = u_1, ..., u_k$ is called a *critical chain* if and only if for $1 \leq i < k$, u_{i+1} is a critical predecessor of u_i. The *head* of the chain is u_1 and the tail is u_k.

Lemma 3 *For every node $v \in V, C(v)$ is a critical chain.*

Proof. The proof is by induction on the number of iterations of the **while** loop of Algorithm COMPUTE-E-VALUE(v, G). Let C_i be the cluster at iteration i of the loop. Clearly, C_1 is a critical chain since it consists of v and its critical predecessor. Suppose that for some $k \geq 1$, the clusters C_i ($1 \leq i \leq k$) are critical chains. Let C_k be $\{v, w_1, ..., w_k\}$. Therefore, m_k=GREEDY-SCHEDULE ($\{w_1, ..., w_k\}$) and c_k=MAX-C-VALUE($\{v, w_1, ..., w_k\}$). If $m_k \geq c_k$ then the loop is terminated and the algorithm returns a cluster C_j, $j \leq k$, which is a critical chain. Thus the claim holds.

Suppose that $m_k < c_k$. Then the algorithm executes iteration $k+1$. We show that C_{k+1} is also a critical chain. Let (x, w_j) be an arc with maximum c value that crosses C_{k+1}; thus $c(x, w_j) = c_k$ and $j \in \{0, ..., k\}$. If $j = k$ then x is a critical predecessor of w_j. Therefore $C_{k+1} = C_k \cup \{x\}$ is a critical chain, and the claim holds.

So suppose that $j < k$. It follows that $m_k < c(x, w_j) \leq c(w_{j+1}, w_j)$ (since w_{j+1} is a critical predecessor of w_j). But:

$$\begin{aligned} m_k &\geq e(w_{j+1}) + \mu(w_{j+1}) + \mu(w_j) \\ &\geq e(w_{j+1}) + \mu(w_{j+1}) + \lambda(w_{j+1}, w_j), \text{ since } g(w_{j+1}) \geq 1 \\ &= c(w_{j+1}, w_j) \\ &> m_k, \text{ a contradiction.} \end{aligned}$$

Therefore, x must be a critical predecessor of w_k and hence C_{k+1} is a critical chain.

Consider two critical chains C_1 and C_2 such that $tail(C_1) = head(C_2)$. We define $C_1 \oplus C_2$ as the critical chain that results when $tail(C_1)$ is replaced by C_2. Finally, for $v \in V$, let $C^*(v)$ be the critical chain returned by the following steps:

repeat

let $w = tail(C(v))$;

$C(v) \leftarrow C(v) \oplus C(w)$;

until $C(w) = \{w\}$.

To construct a clustering for G, we proceed as before: i.e., we begin by computing the e values and the clusters of the nodes of G using Algorithm COMPUTE-E-VALUE. Next, we compute $\Phi(G)$ except that now we add C^* to $\Phi(G)$ (instead of $C(v)$). We now prove the following.

Theorem 3 *Let v be mapped to processor p and let its start time on p be $s(v,p)$. Then $s(v,p) = e(v)$.*

Proof. The theorem is trivially true for all source nodes. Let v be a non-source node and assume that all of its immediate predecessors have start times equal to their e values. Suppose that v resides in critical chain C^*. We have two cases:

Case 1 : $v \neq tail(C^*)$. Let u be the critical predecessor of v in C^*. Then

$s(v,p)$=max$\{e(u)+\mu(u)$,max$\{c(w,v)|(w,v)\in E$ and $w\notin C^*\}\}\leq e(v)$.

Case 2 : $v = tail(C^*)$. By definition of C^*, it should be the case that $C(v) = \{v\}$. Hence, $s(v,p)$ =max$\{c(w,v)\in E\} = e(v)$.

3.4.2 Trees

An *intree* is a directed rooted tree in which every arc is directed from a node to its parent. An *outtree* is similar except that every arc is directed from a node to its children. If duplication is allowed, the task clustering problem for outtrees can be solved optimally using the following simple algorithm [7]: map every root-to-leaf path to a processor and execute each node as its e value. If duplication is not allowed, the task clustering problem for outtrees is NP-complete [7]. For the case of intrees, it was shown in [7] that the task clustering problem with no task duplication is also NP-complete. Moreover, allowing duplication does not help because duplication tasks can always be removed without increasing the makespan.

An interesting question is whether there are good approximation algorithms for scheduling intrees and outtrees when no duplication is allowed. The answer is affirmative: our greedy algorithm can be construct 2-optimal schedules for both intrees and outtrees.

Corollary 1 *When duplication is not allowed, there is a polynomial-time algorithm that constructs a clustering for an intree with granularity at least* $(1-\varepsilon)/\varepsilon$ *whose makespan is at most* $(1+\varepsilon)$ *times the makespan of an optimal clustering.*

Proof. Let v be a node in a DAG G. It is easy to verify that for the clustering produced by our algorithm, a necessary condition for v to have duplicates is that it has two descendants u_1 and u_2 such that one of these nodes, say u_1, is reachable from v via some path that does not contain u_2. If G is an intree, this condition is never true. Thus, the clustering produced by our algorithm contains no duplicate nodes and the result follows.

Corollary 2 *When duplication is not allowed, there is a polynomial-time algorithm that constructs a clustering for an outtree with granularity at least* $(1-\varepsilon)/\varepsilon$ *whose makespan is at most* $(1+\varepsilon)$ *times the makespan of an optimal clustering.*

Proof. Let T be an outtree. T can be converted into an intree T' by reversing the direction of every arc in T. Moreover, the granularities of T and T' are equal. Given a schedule S' for T' (with no task duplication), a schedule S for T with the same makespan can be derived by defining the start time of node v in S as makespan(S')-finish time of v in S' [7]. A similar transformation can be made from any schedule for T to a schedule for T'. Therefore, for an outtree with granularity at least $(1-\varepsilon)/\varepsilon$, a $(1+\varepsilon)$-optimal can be obtained by first converting it into an intree T', then computing a $(1+\varepsilon)$-optimal clustering for T'.

Corollary 3 *When duplication is not allowed, there are polynomial-time algorithms that construct optimal clusterings for* coarse grain *intrees and outtrees.*

Proof. Follows from the fact that our greedy algorithm, when applied to coarse grain DAGs, produces schedules with optimal makespans.

3.5 Summary

In this chapter, we present a new task clustering algorithm that runs $|V|$ times faster than the PY algorithm. Unlike the PY algorithm, the new algorithm uses a simple greedy strategy to find the best cluster for a node v: it maintains only one candidate cluster and "grows" the cluster a node at a time if doing so can potentially decrease the start time of v. In addition, we prove a better performance guarantee by explicitly taking into account the *granularity* of the DAG. We show that if $g(G) \geq (1-\varepsilon)/\varepsilon$ for some $0 < \varepsilon \leq 1$, our algorithm produces a schedule whose makespan is at most $(1+\varepsilon)$ times the optimal makespan. As a corollary, for a DAG with *arbitrary*

granularity (i.e., $g(G) \geq 0$), the algorithm produces a schedule that is at most twice optimal, thus matching the bound of the PY algorithm. However, as $g(G)$ increases, the bound gets better. For example, if $g(G) \geq \frac{1}{2}$ the makespan is at most $\frac{5}{3}$ times optimal.

For coarse grain DAGs (i.e., DAGs whose granularity is at least 1), the task clustering algorithm gives 1.5-optimal schedules. We improve this result by giving a slightly different algorithm that produces *optimal* schedules for coarse grain DAGs.

Finally, we show that the algorithm can be used to solve the task clustering problem with *no* task duplication for directed rooted trees. In particular, we exhibit: (1) 2-optimal schedules for general directed rooted trees; and (2) optimal schedules for coarse grain directed rooted trees. These results are interesting because it is known that task clustering with no duplication is NP-hard even when restricted to directed rooted trees [7].

CHAPTER 4

CLUSTERING STATIC TASK GRAPHS WITHOUT DUPLICATION

For task clustering with no duplication, the DSC algorithm of Gerasoulis and Yang is empirically the best known algorithm to date in terms of both speed and solution quality. The DSC algorithm is based on the critical path method. At each refinement step, it computes the critical path of the clustered DAG constructed so far, i.e., the longest path from a source node to a sink node. (The length of a path is the sum of the node weights and edge weights along the path.) It then zeroes out an edge along the critical path if doing so will decrease the critical path length. The main source of complexity in the DSC algorithm is the computation of the critical path, which is done at each refinement step. On the other hand, the use of critical path information is also the reason why DSC performs very well compared to other algorithms.

In this chapter, we present an algorithm called CASS-II for task clustering with no duplication which is competitive to DSC in terms of both speed and solution quality. With respect to speed, CASS-II is better than DSC: it has a time complexity of $O(|E|+|V|\lg|V|)$, as opposed to DSC's $O((|E|+|V|)\lg|V|)$. Indeed, experimental results (described later in Chapter 5) show that CASS-II is between 3 to 5 times faster than DSC. (It is worth pointing out that we used the C code for DSC developed by the authors of the DSC algorithm. The C code for CASS-II was developed by the author of this thesis.) With respect to solution quality, experimental results show that CASS-II is virtually as good as DSC and, in fact, even outperforms DSC for very fine grain DAGs (granularity less than 0.1).

4.1 CASS-II Algorithm

CASS-II employs a two-step approach. Let $G = (V, E, \mu, \lambda)$ be a weighted DAG. In the first step, CASS-II computes for each node v a value $s(v)$, which is the length of a longest path from a source node to v (excluding the execution time of v). Thus, $s(v)$ is the start time of v prior to any clustering of G. The s values are computed in node topological order of G. The s value of every source node is zero. Let v be a node all of whose immediate predecessors have been assigned s values. Then,

$$s(v) = \max\{s(u) + \mu(u) + \lambda(u, v) | (u, v) \in E\} \tag{4.1}$$

The second step is the clustering step. Just like DSC, it consists of a sequence of refinement steps, where each refinement step creates a new cluster or "grows" an existing cluster. Unlike DSC, CASS-II constructs the clusters bottom-up, i.e., starting from the sink nodes. To construct the clusters, the algorithm computes for each node v, a value $f(v)$, which is the longest path from v to a sink node in the current partially clustered DAG. Let $l(v) = s(v) + f(v)$. The algorithm uses $l(v)$ to determine whether the node v can be considered for clustering at current refinement step.

More precisely, the algorithm begins by placing every sink node v in its own cluster and by setting $f(v) = \mu(v)$ (hence, $l(v) = s(v) + \mu(v)$). The algorithm then goes through a sequence of iterations, where at each iteration it considers for clustering every node u all of whose immediate successors have been clustered (and hence been assigned f values). Call such a node *current*. For every current node u, its f value is computed as

$$f(u) = \max\{\mu(u) + \lambda(u, v) + f(v) | (u, v) \in E\} \tag{4.2}$$

The immediate successor v which determines $f(u)$ is defined as the *dominant successor* of current node u. In general, two or more current nodes may share the same dominant successor. Figure 4.1 illustrates the computation of the f values of current nodes. In Figure 4.1, nodes u_1 and u_2 are current nodes, but u_3 is not since one of its immediate successors, u_2, has not been assigned an f value. Since v_1 is the only immediate successor of u_1, $f(u_1) = \mu(u_1) + \lambda(u_1, v_1) + f(v_1) = 32$, and v_1 is the dominant successor of u_1. On the other hand, current node u_2 has two immediate successors v_1 and v_4. Thus, $f(u_2) = \max\{\mu(u_2) + \lambda(u_2, v_1) + f(v_1), \mu(u_2) + \lambda(u_2, v_4) + f(v_4)\} = 26$; the dominant successor of u_2 is v_1. Thus, current nodes u_1 and u_2 have the same dominant successor, v_1. Note that $l(u_1) = s(u_1) + f(u_1) = 18 + 32 = 50$ and $l(u_2) = s(u_2) + f(u_2) = 21 + 26 = 47$. Finally, we define the f value of a cluster as the f value of the first node in the cluster. For example, in Figure 4.1, assuming that node v_1 is the first node in the cluster C_1, then $f(C_1) = f(v_1) = 17$.

Once the l values of all current nodes have been computed, one of them will be placed in a cluster during the current iteration. The current nodes are considered for clustering in nondecreasing order of their l values. For a given current node u, let v be its dominant successor and let C_v be the cluster containing v. Then, u is included in the cluster C_v if doing so does not increase *both* $f(u)$ and $f(C_v)$. Otherwise, u is placed in a new cluster all to itself.

Figure 4.2 illustrates the method that could be used for including a current node u_1 in a cluster C. In the figure, it is assumed that v_1 is the first node in C and v_3 is the first immediate successor of u_1 in C (with respect to the sequential ordering). Then u_1 is included in C by placing it either: (1) immediately before v_1, or (2) immediately before v_3. Note if u_1 is inserted before v_1, $f(u_1)$ may increase but the f values of the other nodes in C would not change. On the other hand, if u_1 is inserted before v_3, $f(u_1)$ may decrease but $f(v_1)$ and $f(v_2)$ may increase.

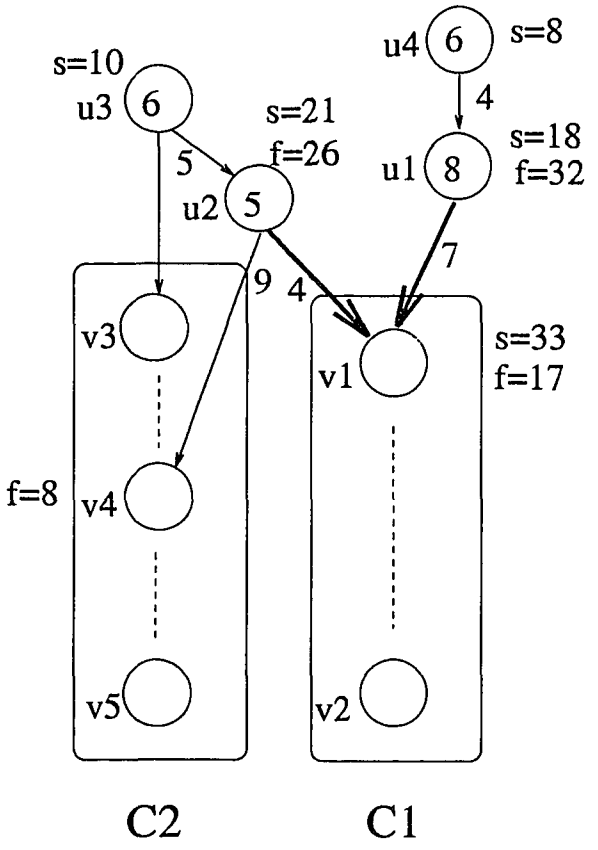


Figure 4.1 An example of computing the f values of current nodes.

In the second case, updating the $f(v_1)$ and $f(v_2)$ will increase the time complexity. Therefore, only the first case is used by CASS-II. Note that we also do not update the l and s values for the nodes in the cluster. If the placement is not *acceptable* (that is, reduce the f values of neither the current node nor the cluster), the current node is placed in a new cluster all to itself.

Figure 4.3 shows the result of applying the clustering strategy to Figure 4.1. Current node u_1 will be considered first since it has a higher l value than current node u_2. If u_1 were included in the cluster C_1 containing its dominant successor v_1, $f(u_1)$ would be reduced from 32 to 25 and $f(C_1)$ would not be changed. (For C_1, its f value would be determined by node v_1, whose f value would remain 17 but whose s value would now be $s(u_2) + \mu(u_2) + \lambda(u_2, v_1) = 21 + 5 + 4 = 30$. Hence,

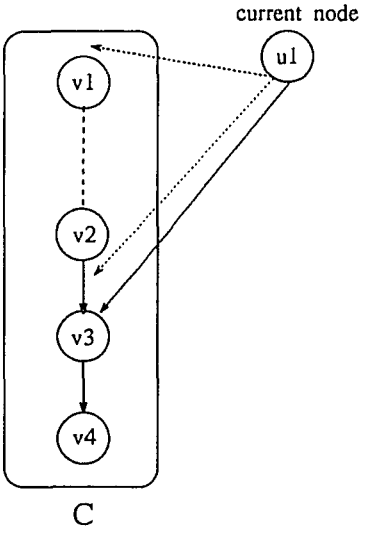


Figure 4.2 The strategy of edge zeroing.

$l(v_1) = 47$.) Thus, the clustering is acceptable and current node u_1 is included in cluster C_1. Next, current node u_2 is considered. One can check that clustering u_2 with C_1 increases $f(u_2)$; hence u_2 is placed in a new cluster all to itself.

The complete algorithm is given as Algorithm CASS-II below. The algorithm maintains a priority queue S consisting of items of the form $[u, l(u), v]$, where u is a current node, $l(u)$ is the l value of u, and v is the dominant successor of u. INSERT($S, item$) inserts an item in S and DELETE-MAX-L-VALUE(S) deletes from S the item with the maximum l value. The algorithm returns, for each node v, the cluster $C(v)$ containing it.

1. **Algorithm** CASS-II(G)
2. **begin**
3. **for** each node v **do**
4. compute $s(v)$;
5. **endfor**;

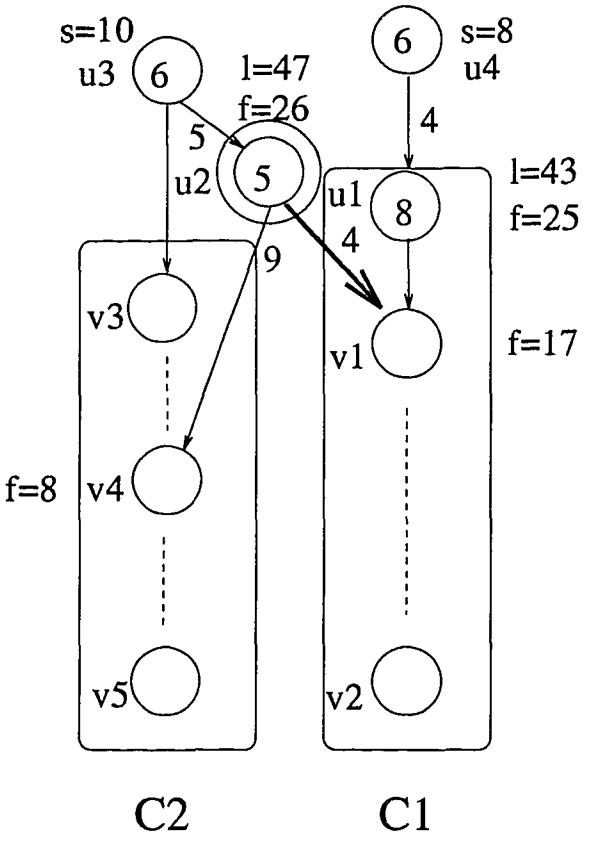


Figure 4.3 The clustering of the DAG in Figure 4.1.

6. **for** each sink node v **do**

7. $f(v) \leftarrow \mu(v)$; $l(v) = s(v) + f(v)$;

8. $C(v) \leftarrow \{v\}$;

9. **endfor**;

10. $S \leftarrow \emptyset$;

11. **while** there are current nodes **do**

12. **for** each new current node u **do**

13. find u's dominant successor v;

14. $f(u) \leftarrow \mu(u) + \lambda(u, v) + f(v)$;

15. $l(u) \leftarrow s(u) + f(u)$;

16. INSERT($S, [u, l(u), v]$);

17. **endfor**;

18. $[x, l(x), y] \leftarrow$ DELETE-MAX-L-VALUE(S);

19. **if** $C(y) \cup x$ is acceptable **then**

20. $C(x) \leftarrow C(y) \leftarrow C(y) \cup \{x\}$;

21. **else**

22. $C(x) \leftarrow \{x\}$;

23. **endif**;

24. **endwhile;**

25. **return** ($\{C(v) | v \in G\}$);

26. **end** CASS-II.

For the DAG of Figure 2.4, Figure 4.4 shows each iteration of Algorithm CASS-II. The figure also shows the l values of the tasks in each step. The makespan of the clustering shown in Figure 4.4(l) is 26. On the other hand, applying DSC on the example DAG results in the following clusters:

$$C_1 = \{1\},\ C_2 = \{2, 5\},\ C_3 = \{3, 6, 10\},$$
$$C_4 = \{4, 7, 9\},\ C_5 = \{8\}.$$

The makespan of the clustering produced by the DSC algorithm is 30 by searching both directions (i.e., top-down and bottom-up) of the DAG. Note that in practice, like the DSC algorithm, CASS-II also searches both directions of the DAG and uses the makespan of whichever is better.

4.2 Complexity Analysis

We now analyze the complexity of Algorithm CASS-II. The s values of all nodes (lines 3-5) can be computed in time $O(|V| + |E|)$. Initializing the sink nodes (lines 6-10) takes $O(|V|)$ time. Each iteration of the main **while** loop (lines 11-26) consists of: (1) identifying the current nodes; (2) for each current node, determining its dominant successor, computing its f and l values, and inserting a corresponding item in the queue S; and (3) for each current node deleted from S, determining the cluster to which it belongs.

The current nodes at the start of each iteration can be determined by simply computing for each node u, a value $num(u)$ which is the number of immediate successors that have not yet been clustered. Initially (prior to any clustering), $num(u)$ is simply the total number of immediate successors of u. Whenever a node, say v, is assigned its cluster, the num value of every immediate predecessor of v is decremented by 1. Thus, at the start of each new iteration, the current nodes are those nodes which have not been clustered and which have num values equal to zero. It is easy to verify that updating the num values and determining the current nodes take $O(|V| + |E|)$ time overall.

The dominant successor of a current node can be determined in a similar way. For each node u, we keep track of the "candidate" dominant successor v, which is the immediate successor of u that has already been clustered and for which $\mu(u) + \lambda(u, v) + f(v)$ is maximum. This latter value will be the "candidate" value for $f(u)$.

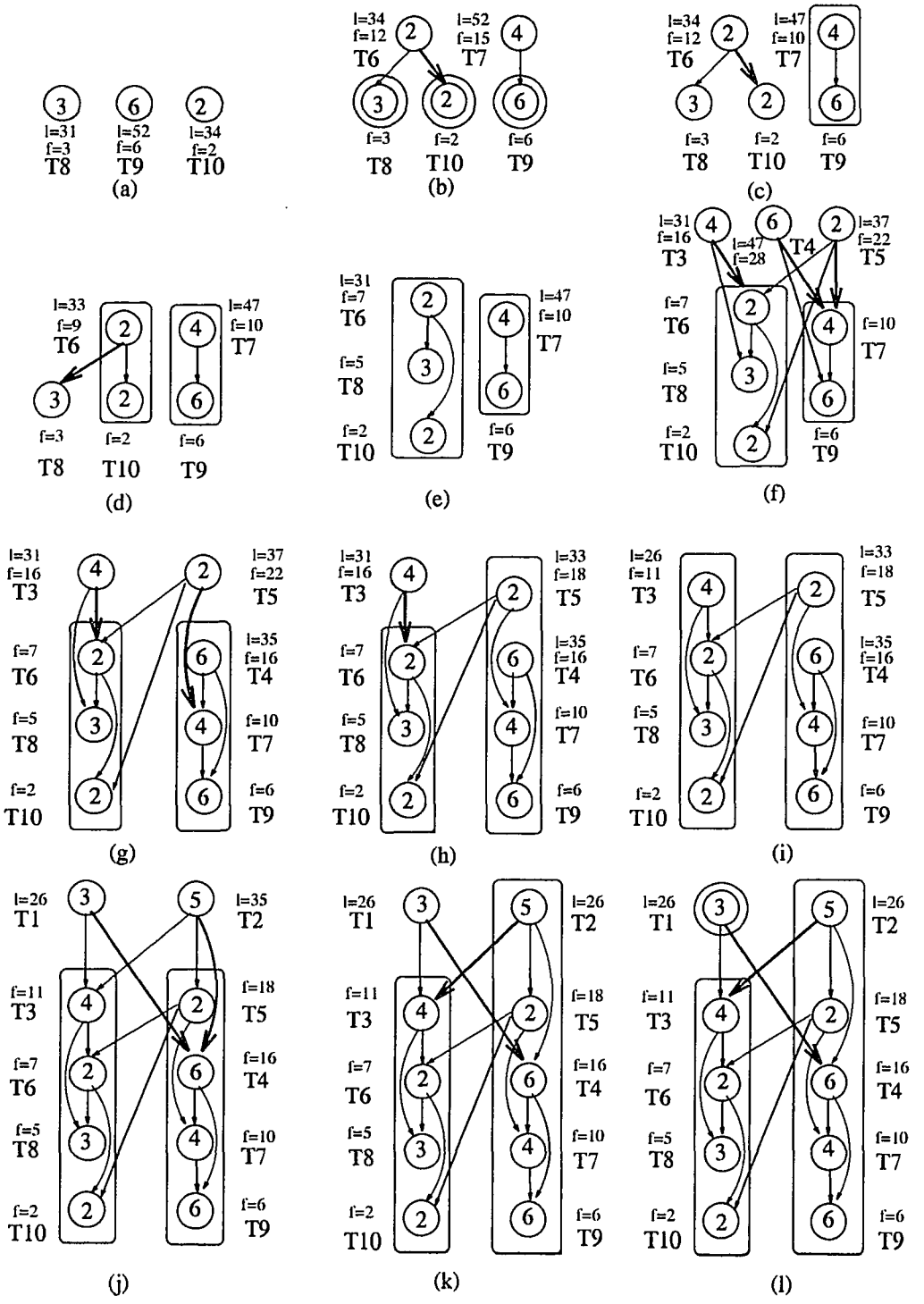


Figure 4.4 The l values and clusters for the DAG of Figure 2.4.

Whenever a node v is assigned its cluster, the candidate f value of every immediate predecessor u of v is updated as $\max\{f(u), \mu(u) + \lambda(u,v) + f(v)\}$. If $f(u)$ changes in value, then the candidate dominant successor of u is changed to v. Thus, when node u becomes a current node, its candidate dominant successor and its candidate f value are the true dominant successor and the true f value, respectively. Overall, keeping track of the candidate values takes $O(|V| + |E|)$ time.

Consider next the insert and delete operations on the priority queue S. By implementing S as a heap, each insert operation and each delete operation takes $O(\lg |S|)$ time. Since there are a total of $|V|$ insert operations and V delete operations, these operations contribute at most $O(|V| \lg |V|)$ to the total time.

Finally, for each current node v, its cluster can be computed in $O(1)$ time (since the s valuses may be changed later, we did not update the l and s values of nodes in the cluster, this reduces the complexity to $O(1)$), thus contributing $O(|V|)$ time overall. It follows that Algorithm CASS-II runs in $O(|E| + |V| \lg |V|)$ time.

4.3 Special Cases

In this section, we discuss the performance of CASS-II for fork and join DAGs, and show the optimality of the algorithm for these DAGs.

Figure 4.5 demonstrates the clustering steps of CASS-II for a fork DAG. Without loss of generality, we assume that the leaf nodes in the DAG shown in (a) are sorted in a nonincreasing order of the c values (i.e., $c(v_1) \geq c(v_2) \geq \cdots \geq c(v_m)$). Initially, each node is in a unit cluster as shown in Figure 4.5(b). The f value of u equals to $\max_i\{c(v_i)\}$ which is $c(v_1)$. At step 2 shown in (c), the cluster of v_1 is grown to include node u. $f(u)$ is reduced to $c(v_2)$ at this moment. The cluster will keep growing until the following condition can not be satisfied.

$$\sum_{i=1}^{k} \mu_i \leq \lambda_{k+1}$$

As shown in (d), CASS-II stops at node v_{k+1} and the original leftmost scheduled cluster forms a linear chain. The steps applied in fork DAGs can be applied to join DAGs by just simply reversing the join DAG into a fork DAG.

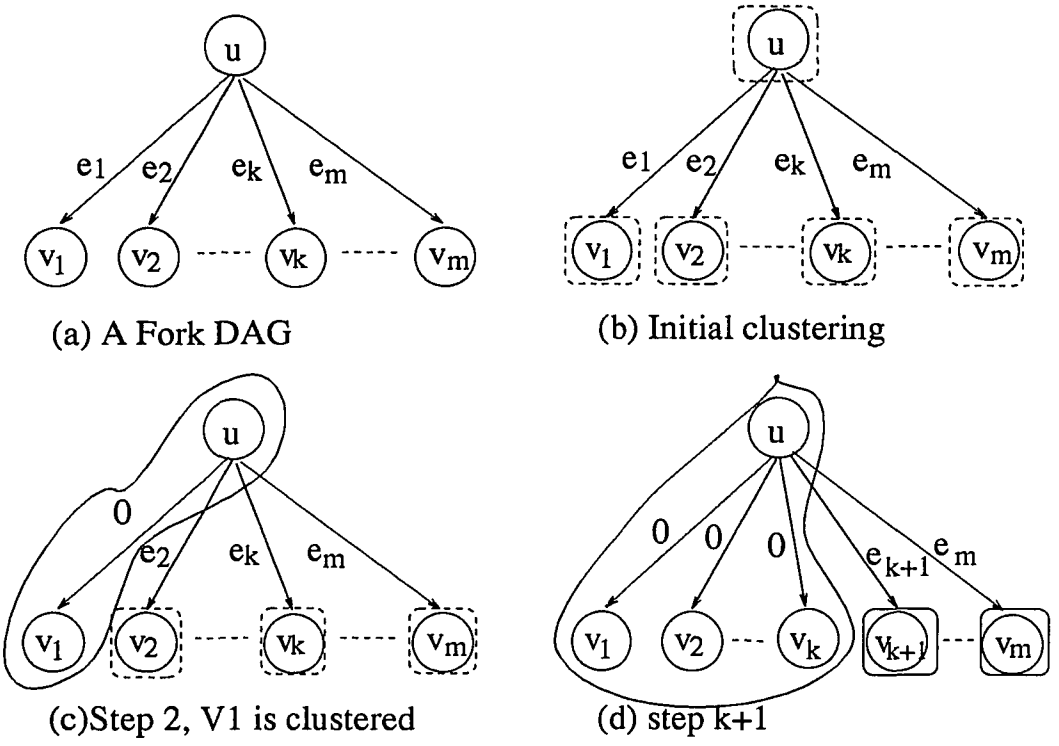


Figure 4.5 CASS-II clustering steps for a fork DAG.

Theorem 4 *CASS-II achieves optimal scheduling for fork and join DAGs.*

Proof: The proof is quite simple. After $f(u)$ is determined. CASS-II will examine free nodes $v_1, v_2, \cdots, v_m$ in a nonincreasing order of c values. Assume the optimal parallel time to be PT_{opt}. Let the optimal scheduling stop at node p, and $t(C) = \sum_{i=1}^{p} \mu(v_i)$. Then the optimal PT is:

$$PT_{opt} = \mu(u) + max(t(C), \mu(v_{p+1}) + \lambda(e_{p+1})) \tag{4.3}$$

CASS-II zeroes edges from left to right, as many as possible, up to the point k as

shown in Figure 4.5(d) such that:

$$\sum_{i=1}^{k-1} \mu_i \leq \lambda_k$$

and

$$\sum_{i=1}^{k} \mu_i \geq \lambda_{k+1} \tag{4.4}$$

Suppose that $p \neq k$ and $PT_{opt} \leq PT_{CASS-II}$. There are two cases:

If $p < k$, then

$$\sum_{i=1}^{p} \mu_i < \sum_{i=1}^{k} \mu_i \leq \mu(v_k) + \lambda(e_k) \leq \mu(v_{p+1}) + \lambda(e_{p+1})$$

Thus Equation 4.3 can be simplified as:

$$PT_{opt} = \mu(u) + \lambda(v_{p+1}) + \mu(v_{p+1}) \geq \mu(u) + \lambda(v_k) + \mu(v_k)$$
$$\geq \mu(u) + max(t(C), \mu(v_{k+1}) + \lambda(e_{k+1})) = PT_{CASS-II}$$

If $p > k$ then since $\sum_{i=1}^{p} \mu_i \geq \sum_{i=1}^{k+1} \mu_i \geq \mu(v_{k+1}) + \lambda(e_{k+1}) \geq \lambda(v_{p+1}) + \mu(v_{p+1})$,
$PT_{opt} = \mu(u) + \sum_{i=1}^{p} \mu_i \geq \mu(u) + max(t(C), \mu(v_{k+1}) + \lambda(e_{k+1})) = PT_{CASS-II}$
There is a contradiction in both cases.

For a join, the optimality can be proved using the same analysis by reversing the DAG and the solution is symmetrical to the optimal result for a fork. ■

4.4 Summary

In this chapter, we have presented a simple task clustering algorithm without task duplication. Unlike the DSC algorithm, CASS-II uses only limited "global" information and does not recompute the critical path in each refinement step. Therefore,

the algorithm runs in $O(|E| + |V|lg|V|)$ which is faster than $O((|V| + |E|)lg|V|)$ of the DSC algorithm. Unfortunately, we are unable to find a provable bound on the solution quality produced by the CASS-II. This is very difficult because it is known that for general task graphs, clustering without task duplication remains NP-hard even when the solution quality is relaxed to be within twice the optimal solution. However, we exhibit optimal schedules for the special cases such as join and fork DAGs.

CHAPTER 5

PERFORMANCE COMPARISON AND EXPERIMENTAL RESULTS

This chapter describes experimental results for the CASS-I and CASS-II algorithms introduced in the previous chapters. CASS-I is compared with the PY algorithm [40], which is in theory the best known algorithm for clustering with task duplication. Similarly, CASS-II is compared with the DSC algorithm [18], which is empirically the best known algorithm for clustering without task duplication. Our experimental results demonstrate that the CASS algorithms compare very favorably with their counterparts, and generally outperform the other algorithms in terms of both speed and solution quality.

5.1 Clustering with Task Duplication

For clustering with task duplication, several algorithms are known, e.g., Kruatrachue and Lewis [26], Chung and Ranka [8], Kwok and Ahmad [28], and Papadimitriou and Yannakakis [40]. Table 5.1 compares these algorithms with CASS-I with respect to theoretical runtime and performance guarantee (if any). All algorithms, except that of Kwok and Ahmad, assume an unbounded number of processors; for Kwok and Ahmad's algorithm, the number of processors, p, is an input parameter. Only the PY algorithm and CASS-I have theoretical guarantees on performance. The PY algorithm guarantees schedules with makespan which are at most 2-optimal. CASS-I gives an even tighter bound on performance: $(1+\frac{1}{1+g})$ times optimal for task graphs of granularity g. Moreover, CASS-I achieves the fastest theoretical runtime among the algorithms.

To validate CASS-I, we tested the algorithm on random DAGs. A random DAG is generated by first randomly generating the number of nodes, then randomly

generating edges between them, and finally assigning random node weights and edge weights. For comparison purposes, we also ran the PY algorithm on the same set of DAGs. Our first experiment consisted on 300 DAGs; the results are summarized in Table 5.2.

ALGORITHM	AUTHORS	PERFORM. GUARANTEE	RUNTIME OF ALGORITHM
Duplication Scheduling Heuristic	Kruatrachue & Lewis, 1987	None	$O(n^4)$
Bottom-Up Top-Down Duplication Heuristic	Chung & Ranka, 1992	None	$O(n^4)$
Critical path Fast Duplication Heuristic	Kwok & Ahmad, 1994	None	$O(n^2cp)$
PY Heuristic	Papadimitriou & Yannakakis, 1990	2-optimal	$O(n^3lgn + n^2e)$
CASS-I Heuristic	Palis, Liou & Wei, 1994	1+1/(1+g) -optimal	$O(n^2lgn + ne)$

Table 5.1 A comparison of clustering algorithms with task duplication. n = no. of tasks. e = no. of edges. p = no. of processors.

	# of DAGs	# nodes Min-Max	M(PY,CASS-I) Avg.	T(PY,CASS-I) Avg.
G1*	100	6-64	1.23	1.13
G2	100	64-256	1.30	1.72
G3	100	256-511	1.36	1.72

Table 5.2 Experimental results for CASS-I and PY run on a 386PC* and a DEC5900.

The 300 DAGs are divided into three groups of 100 DAGs each. Group G1 was tested on CASS-I and PY running on a 386 PC; groups G2 and G3 were tested on the algorithms running on a DEC 5900. Column 3 of the Table 5.2 gives the range of number of nodes for the DAGs in the group. Column 4 gives the average

makespan ratio of PY over CASS-I. In general, given two clustering algorithms A and B, the *average makespan ratio of A over B*, M(A,B), is defined as

$$M(A,B) = \frac{1}{N}\sum_{i=1}^{N}\frac{makespan_A(G_i)}{makespan_B(G_i)}$$

where N is the number of DAGs. Finally, column 4 gives the average runtime ratio of PY over CASS-I. Given two clustering algorithms A and B, the *average runtime ratio of A over B*, T(A,B), is defined as

$$T(A,B) = \frac{1}{N}\sum_{i=1}^{N}\frac{runtime_A(G_i)}{runtime_B(G_i)}$$

Table 5.2 indicates that CASS-I gives 23%-36% shorter makespans than PY, in 13%-72% less time.

To determine how well both algorithms perform on DAGs with varying grain size, we conducted another experiment on 280 DAGs, where they are divided into 14 groups, as shown in Table 5.3. Column 1 of the table is interpreted as follows: grain size = 0.1 means a group of 20 DAGs with granularity in the range (0, 0.1], grain size = 0.2 means 20 DAGs with granularity in the range (0.1, 0.2], and so on. Column 2 gives the range of number of nodes for DAGs in the group. Column 3 and 4 give the average runtime (in seconds) for PY and CASS-I, respectively, running on a Sun Sparc workstation. Column 5 is the average makespan ratio of PY over the *lower bound* (LB) on the optimal makespan (obtained from the e values computed by CASS-I). Finally, Column 6 is the average makespan ratio of CASS-I over the lower bound on the optimal makespan.

For ease of comparison, Table 5.4 shows the average makespan ratio and average runtime ratio of PY over CASS-I.

Grain Size	# of Tasks Min - Max	PY Runtime (sec)	CASS-I Runtime (sec)	M(PY,LB)	M(CASS-I,LB)
0.1	91-996	10.94	6.22	1.80	1.90
0.2	71-910	3.68	2.45	1.70	1.74
0.3	137-1009	2.85	2.09	1.58	1.58
0.4	84-989	1.97	1.57	1.53	1.48
0.5	97-1022	1.60	1.23	1.45	1.37
0.6	124-1002	1.36	1.16	1.45	1.40
0.7	77-982	1.40	1.01	1.41	1.33
0.8	90-1015	1.12	1.13	1.36	1.30
0.9	123-962	1.44	1.02	1.33	1.24
1.0	70-942	1.36	1.06	1.31	1.22
2.0	202-889	1.30	1.19	1.27	1.27
3.0	83-955	1.29	0.89	1.16	1.14
4.0	129-935	1.13	1.01	1.11	1.09
5.0	76-882	1.01	0.91	1.09	1.04

Table 5.3 Experimental results for CASS-I and PY run on a Sun Sparc workstation.

Table 5.4 reveals that there are instances where PY gives better makespans than CASS-I (grain size = 0.1 and 0.2), although for the majority of the cases, CASS-I is better. CASS-I is also significantly faster than PY (up to 76% faster for grain size = 0.1), although there is one surprising instance (grain size = 0.8) for which PY is slightly faster.

Finally, Figure 5.1 plots the average makespan ratio of PY and CASS-I over the lower bound on the optimal makespan (column 5 and 6 of Table 5.3). The plot labeled "performance bound" is $(1+\frac{1}{1+g})$, where g is the granularity. It validates the theoretical guarantee on performance for CASS-I; i.e., the makespans generated by CASS-I are within $(1+\frac{1}{1+g})$ times the optimal makespans. Note that the experimental results are pessimistic, because we used the *lower bound* on the optimal makespan, instead of the optimal makespan (which we are unable to compute). Surprisingly, the graph indicates that the PY algorithm also satisfies the same performance upper bound, at least empirically. We conjecture that a tighter analysis of the PY algorithm

Grain Size	M(PY,CASS-I)	T(PY,CASS-I)
0.1	0.95	1.76
0.2	0.97	1.50
0.3	1.00	1.36
0.4	1.03	1.26
0.5	1.05	1.30
0.6	1.03	1.17
0.7	1.06	1.38
0.8	1.05	0.99
0.9	1.08	1.41
1.0	1.08	1.28
2.0	1.00	1.10
3.0	1.02	1.44
4.0	1.03	1.12
5.0	1.04	1.11

Table 5.4 The average makespan ratio and average runtime ratio of PY over CASS-I.

(i.e., by taking task graph granularity into account) would likewise prove that it satisfies the performance upper bound in theory.

5.2 Clustering without Task Duplication

For clustering without task duplication, a number of other algorithms have been reported in the literature besides DSC and CASS-II. The most well-known are the algorithms proposed by Sarkar [43], the MCP heuristic of Wu and Gajski [49], and the algorithm of Kim and Browne [24]. All algorithms assume an unbounded number of processors. Table 5.5 compares these algorithms with CASS-II with respect to performance guarantee and theoretical runtime. It shows that none of the algorithms have a performance guarantee on general task graphs. Among these algorithms, CASS-II is superior to the others in terms of speed.

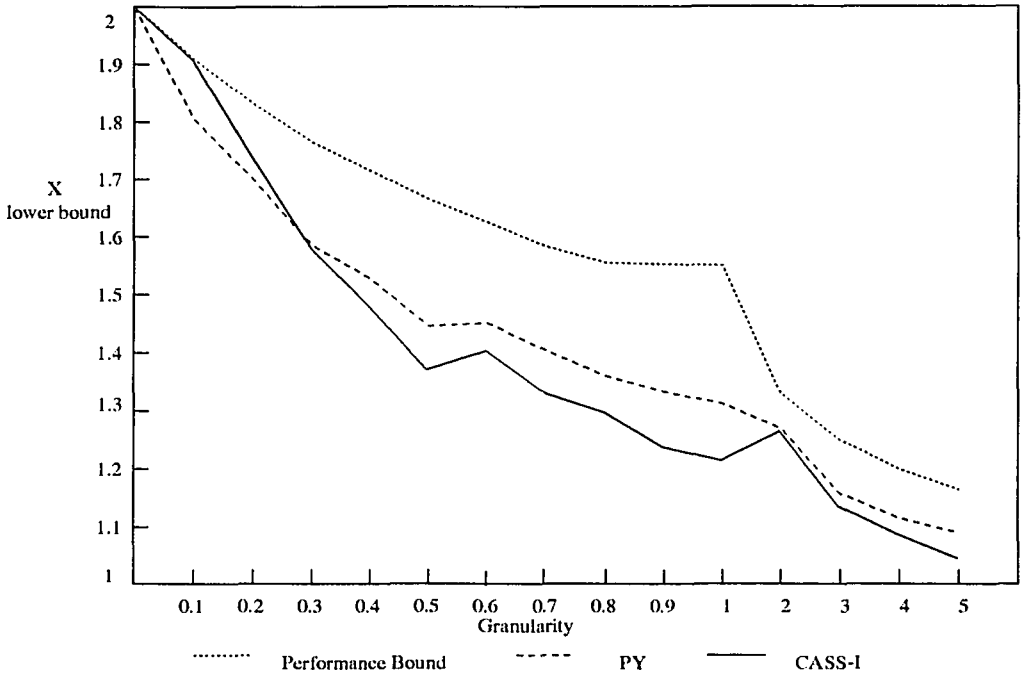


Figure 5.1 Average makespan ratio of PY and CASS-I over the lower bound on optimal makespan.

	Sarkar	MCP	Kim & Browne	DSC	CASS-II
Join/Fork	no	no	no	optimal	optimal
General DAGs	no	no	no	no	no
Runtime	$O(e(n+e))$	$O(n^2lgn)$	$O(n(n+e))$	$O((n+e)lgn)$	$O(e+nlgn)$

Table 5.5 A comparison of static clustering algorithms without task duplication. n = no. of tasks. e = no. of edges.

Experimental results in [16, 18, 51] have shown that DSC outperforms all the algorithms listed in Table 5.5, except for CASS-II, in terms of both speed and solution quality. To see how CASS-II compares with DSC, we tested both algorithms on 350 DAGs. The 350 DAGs were divided into 14 groups of 25 DAGs each according to their grain size, as indicated by column 1 of Table 5.6. Column 2 of the table gives the range of number of nodes for the DAGs in the group. Column 3 and 4 give the average runtimes (in milliseconds) of DSC and CASS-II, respectively, when executed on a Sun Sparc workstation. Column 5 gives the average makespan ratio of DSC

over CASS-II. Finally, column 6 gives the average runtime ratio of DSC over CASS-II.

Grain Size	# of Tasks Min - Max	DSC Avg Runtime	CASS-II's Avg Runtime	M(DSC,CASS-II)	T(DSC,CASS-II)
0.1	85-988	544	131	1.37	4.15
0.2	129-987	698	156	1.10	4.47
0.3	86-988	684	149	1.05	4.59
0.4	87-989	566	129	1.00	4.39
0.5	88-990	532	123	1.01	4.33
0.6	89-949	533	116	1.00	4.59
0.7	90-997	615	141	1.00	4.36
0.8	89-992	524	136	0.99	3.85
0.9	93-993	652	141	0.98	4.62
1.0	90-992	687	133	0.97	5.17
2.0	91-993	930	178	1.00	5.22
3.0	92-994	884	171	1.00	5.17
4.0	93-953	851	159	1.00	5.35
5.0	94-995	737	155	1.00	4.75

Table 5.6 Experimental results of CASS-II and DSC algorithm run on a Sun Sparc workstation.

Table 5.6 indicates that CASS-II is between 3.85 to 5.35 times faster than DSC. Moreover, in terms of solution quality, CASS-II is very competitive: it is better than DSC for grain sizes less or equal to 0.6, and its superiority increases as the DAG becomes increasingly fine grain. For example, for grain sizes equal to 0.1 or less, CASS-II generates makespans which are up to 37% shorter than DSC's. For some DAGs with this grain size, sequentializing a set of tasks (one cluster produced) can achieve a better makespan than executing them in parallel (two or more clusters produced). It is interesting that the case can be detected by CASS-II, but DSC seems not to find the necessity for serial execution and produces several clusters instead of one. On the other hand, for task graph with grain size 0.6 or greater, DSC becomes competitive and in fact even outperforms CASS-II, although by no more than 3% (grain size = 1.0).

5.3 Summary

Our experimental results validate the theoretical guarantee on the performance of CASS-I, i.e., it generates schedules whose makespans are at most $(1 + \frac{1}{1+g})$ times the optimal makespan. We have also compared CASS-I with the PY algorithm and demonstrated that CASS-I generally outperforms PY in terms of both speed and solution quality.

We have also compared CASS-II with the DSC algorithm and showed that CASS-II is 3 to 5 times faster than DSC. For fine grain DAGs (granularity = 0.6 or less), CASS-II consistently gives better schedules. For DAGs with grain size 0.6 or greater, DSC becomes comparable to CASS-II, and in some cases even strictly better, but by no more than 3%.

In summary, our experimental results demonstrate that both CASS-I (for clustering with task duplication) and CASS-II (for clustering without task duplication) are very competitive algorithms in terms of speed and solution quality, outperforming the best currently known algorithms (PY for clustering with task duplication and DSC for clustering without task duplication).

CHAPTER 6

SCHEDULING OF CLUSTERS ON PHYSICAL PROCESSORS

The role of the CASS scheduling module is to merge the task clusters generated by the CASS clustering module onto a fixed number of processors and to determine the order of execution of tasks within each processor. The scheduling module consists of the following sequence of optimization steps:

1. *Cluster Merging* : given m task clusters and n processors such that $m > n$, merge the clusters so that the number of remaining clusters equals to n.

2. *Processor Assignment* : given n clusters and n processors, find a one-to-one mapping of the clusters to the processors taking into account the underlying network topology.

3. *Local Scheduling* : determine the order of execution of tasks mapped to the same processor.

6.1 Cluster Merging

The cluster merging step is performed whenever the number of task clusters is greater than the number of physical processors. We have investigated three approaches for cluster merging: (a) load balancing (LB), (b) communication traffic minimizing (CTM), and (c) random (RAND). All three approaches execute a sequence of refinement steps; each refinement step reduces the number of clusters by one by *merging* a pair of clusters into a single cluster. The approaches essentially differ in their choice of the pair of clusters to be merged:

- *LB*

  Define the (computational) *workload* of a cluster as the sum of execution times

of the tasks in the cluster. At each refinement step, choose a cluster, C_1, that has a minimum workload among all clusters, and find a cluster, C_2, that has a minimum workload among those clusters which have communication edge between it and C_1. Then the pair of clusters C_1 and C_2 are merged.

- *CTM*

  Define the (communication) traffic of a pair of clusters (C_1, C_2) as the sum of communication times of the edges from C_1 to C_2 and from C_2 to C_1. At each refinement step, merge the pair of clusters which has the most traffic.

- *RAND*

  At each refinement step, merge a random pair of clusters.

LB merges the clusters to processors so that the processors have approximately equal workload; i.e., they spend approximately the same time executing tasks, ignoring the inter-cluster communication delays. In CTM, the clusters are merged to the processors so that the total inter-cluster communication is minimized; the workload of the processors is ignored. Finally, RAND ignores both the workload and the communication traffic when mapping the clusters to the processors.

Consider, for example, the clustered task graph (with task duplication) shown in Figure 3.4. When the 5 task clusters are merged to the physical processors, LB and CTM result in the same two clusters, as shown in Figure 6.1. On the other hand, consider the clustered task graph (without task duplication) shown in Figure 4.4(l). When mapped to two physical processors, LB produces the two clusters shown in Figure 6.2(a) while CTM produces the two clusters shown in Figure 6.2(b). Observe that the clusters produced by LB has a schedule of length 26 while that of CTM has a schedule of length 40, which is significantly longer. As discussed later in this chapter, our experimental results indicate that LB is generally better than CTM, and that RAND is the worst approach.

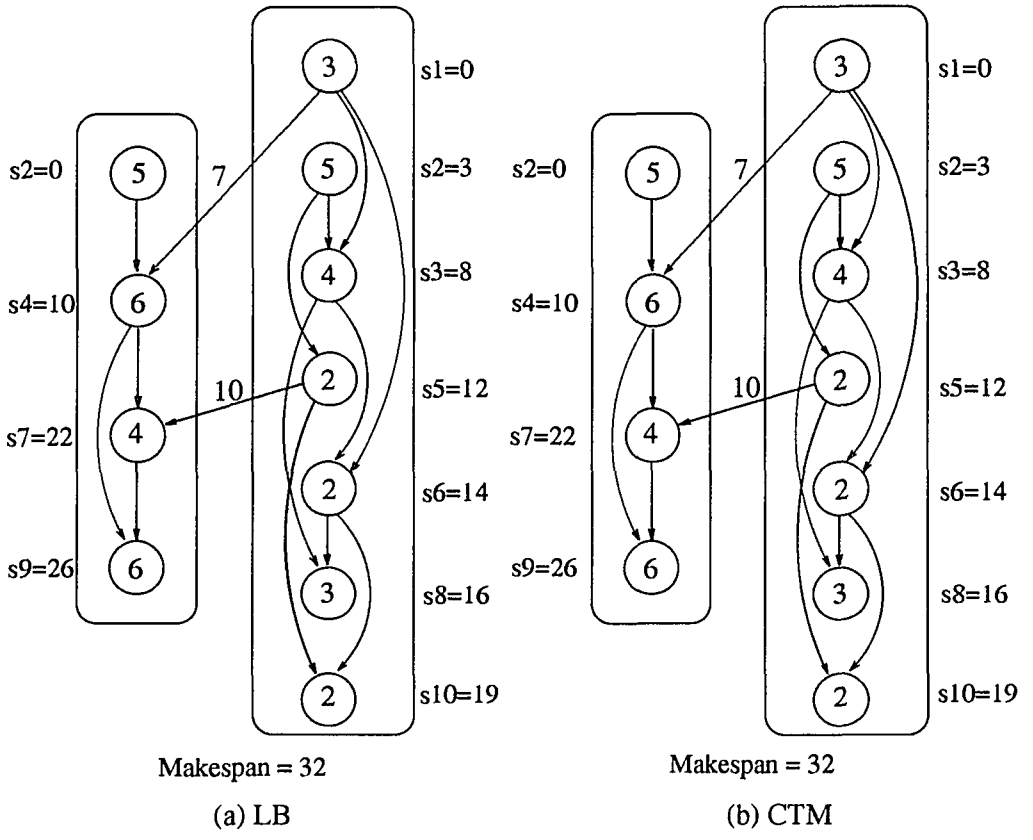


Figure 6.1 The cluster merging of the clustering in Figure 3.4.

6.2 Processor Assignment and Local Scheduling

The processor assignment finds a one-to-one mapping of the clusters to the processors, taking into account the underlying network topology. Presently, CASS uses a simple heuristic to map the cluster to processors: (1) assign to a processor the cluster with the largest total communication traffic with all other clusters; (2) choose an unassigned cluster with the largest communication traffic with an assigned cluster and place it in a processor closest to its communicating partner; (3) repeat (2) until all clusters have been assigned to processors.

For example, consider the situation depicted in Figure 6.3, where four task clusters are to be assigned to four physical processors connected as a linear array. The width of the edges indicates the relative amount of communication traffic between

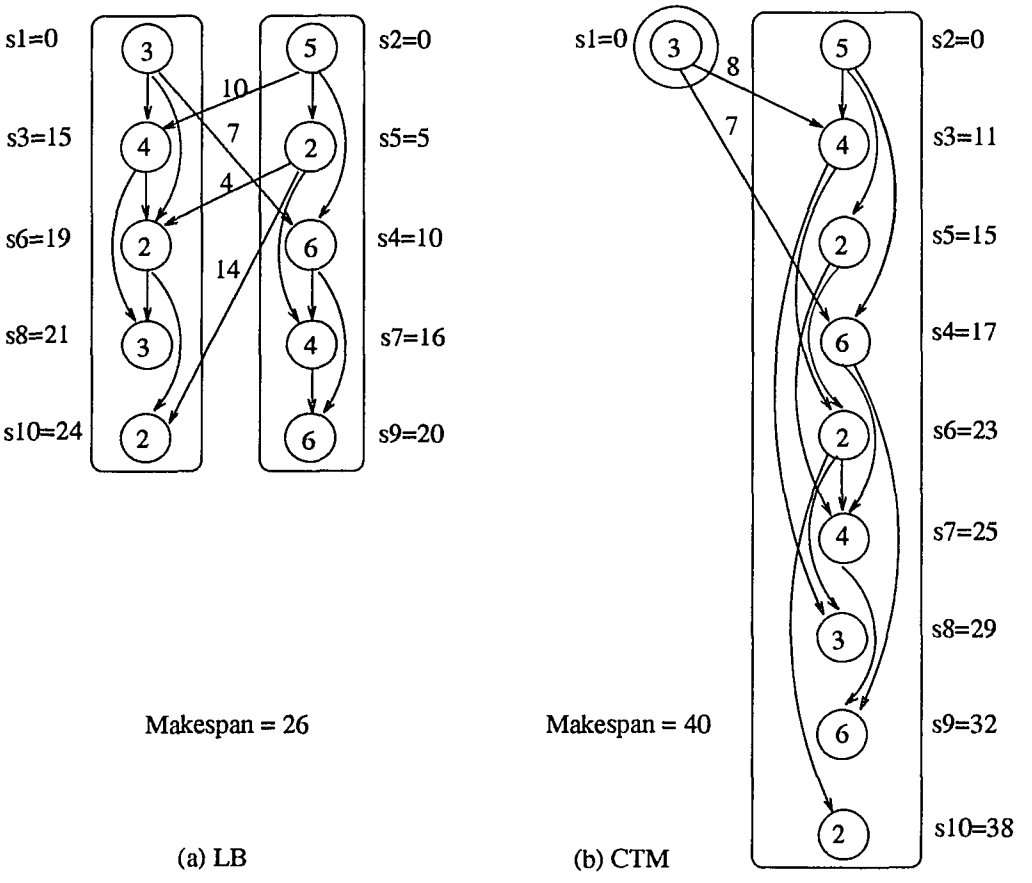


Figure 6.2 The cluster merging of the clustering in Figure 4.4(l).

the clusters. Thus, cluster C_4 communicates most heavily with all other clusters. Consequently, we first assign C_4 to a processor, say the third processor. Next, we consider the cluster which communicates most heavily with C_4; this is C_2. Thus, C_2 is placed on a processor closest to C_4, say the second processor. Then, cluster C_3 is considered and placed on the fourth processor, closest to C_4, which is its most heavily communicating partner. Finally, cluster C_1 is placed on the remaining processor.

Recall that the CASS clustering module specifies, for each task cluster, a sequential order of execution of tasks within the cluster. The cluster merging step – if invoked – also maintains a task execution order for each output cluster (which might be the result of merging some original clusters). This order of execution is derived from the predicted start times of the tasks. Unfortunately, these start times

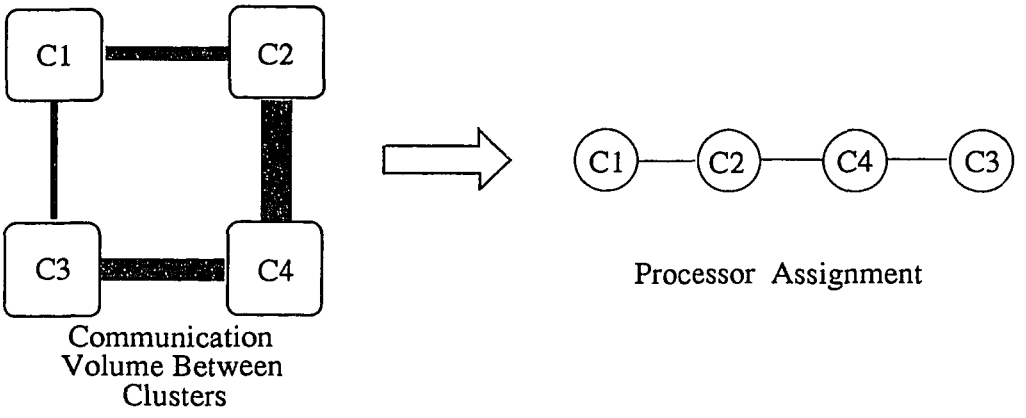


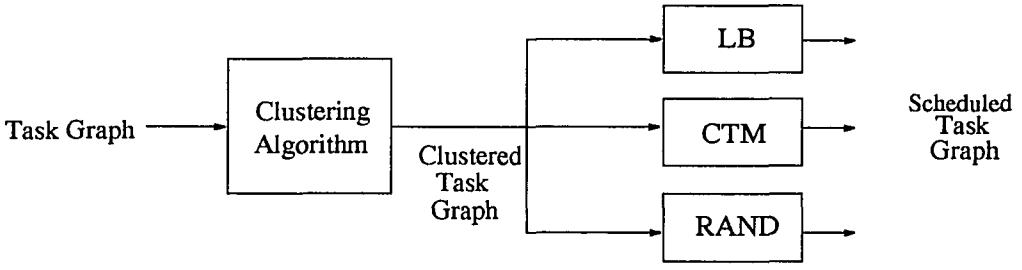
Figure 6.3 An example of processor assignment.

are generally optimistic estimates because they do not take into account the increase in communication delay between tasks that are mapped to non-adjacent processors by the processor assignment step. Thus, it is possible for two independent tasks, say A and B, to have start times $s(A) < s(B)$ before processor assignment and have start times $s'(A) > s'(B)$ after processor assignment because A needs data from a task mapped to a distant processor. The role of local scheduling is to re-order the execution of the tasks to minimize processor idle time, while respecting the precedence constraints between tasks. CASS uses a simple greedy algorithm that maintains a global clock and, at each clock tick, dispatches a task for execution once it has received all the messages it requires. This algorithm is an adaptation of the well-known optimal algorithm for one-processor scheduling for tasks with release times.

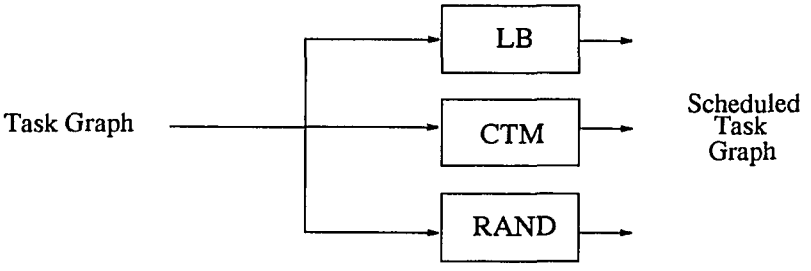
6.3 Experimental Results

We compared the performance of the cluster merging algorithms empirically. Our experimental set-up is shown in Figure 6.4. In the first experiment (the two-phase method), we first applied the CASS-II clustering algorithm to a task graph to produce a clustered task graph. The clustered task graph was then used as input

to the three cluster merging algorithms. In the second experiment (the one-phase method), the clustering algorithm was not applied; instead, the original task graph was used directly by the cluster merging algorithms. In each experiment, we varied the granularity of the original task. Our experimental set-up allows us to compare the performance of the cluster merging algorithms for varying task graph granularity, and with or without prior task clustering. In particular, task clustering, in effect, changes the granularity of the task graph that is input to the cluster merging algorithms. Consequently, the best cluster merging algorithm for the one-phase method may not be the best algorithm for the two-phase method.



(a) Two-Phase Method.



(b) One-Phase Method.

Figure 6.4 Experimental set-up for the cluster merging algorithms.

Our experimental set-up also allows us to assess the effectiveness of task clustering as an intermediate step in the scheduling process. In other words, we

want to determine whether significantly better schedules can be obtained when task clustering is applied, as opposed to not applying it.

We conducted the two experiment on 130 random DAGs. For each DAG, we applied the CASS-II clustering algorithm to produce a clustered DAG with, say, k clusters. We used the integer k to determine the number of processors, m, to use with the cluster merging algorithms. Specifically, for a given k, we choose m to be some power of $2 < k$. We then ran the cluster merging algorithms both on the original DAG (one-phase method) and on the clustered DAG (two-phase method) and compared the makespans of the resulting scheduled DAGs.

Grain Size	Two-Phase			One-Phase		
	LB	CTM	RAND	LB	CTM	RAND
0.1	1.06	1.09	1.11	1.81	1.14	1.86
0.2	0.99	1.00	1.02	1.20	1.10	1.30
0.3	0.97	0.90	0.96	1.25	1.10	1.29
0.4	1.02	0.96	1.02	1.35	1.13	1.34
0.5	1.05	1.06	1.05	1.48	1.21	1.45
0.6	1.10	1.16	1.15	1.58	1.38	1.55
0.7	1.11	1.28	1.24	1.39	1.33	1.45
0.8	1.13	1.32	1.34	1.40	1.50	1.43
0.9	1.14	1.44	1.47	1.39	1.70	1.53
1.0	1.19	1.55	1.62	1.44	1.72	1.51
2.0	1.07	1.09	1.09	1.29	1.39	1.24
3.0	1.12	1.19	1.21	1.27	1.37	1.27
4.0	1.09	1.19	1.22	1.18	1.30	1.25

Table 6.1 Average makespan ratios of cluster merging algorithms (relative to CASS-II) for two-phase and one-phase methods.

Table 6.1 summarizes our experimental results. The first column represents the grain size of the original DAG. Each of the remaining columns gives the average makespan ratio relative to CASS-II, i.e., the average ratio of the makespan produced by the cluster merging algorithm over the makespan produced by the

CASS-II clustering algorithm. It is clear from the table that for a fixed cluster merging algorithm (LB, CTM, RAND), the two-phase method (with prior task clustering) produces significantly better makespans than the one-phase method (no prior task clustering). On average, $LB_{two-phase}$ is better than $LB_{one-phase}$ by 28%, $CTM_{two-phase}$ is better than $CTM_{one-phase}$ by 15%, and $RAND_{two-phase}$ is better than $RAND_{one-phase}$ by 22%.

We should point out that the above results are actually skewed in favor of the one-phase method. The reason is that the number of processors, m, was always chosen to be less than the number of clusters, k, produced by the CASS-II clustering algorithm. In practice, m may be greater than k. If this were the case, then the two-phase method would not perform the cluster merging step (because there are already less clusters than processors) and hence would produce a schedule using k processors and whose makespan equals that produced by the clustering algorithm. On the other hand, the one-phase method would schedule the original task graph consisting of $n > m$ nodes onto m processors using either LB, CTM, or RAND. The resulting schedule would oftentimes be very bad because it still uses way too many processors than necessary. (Recall that the number of clusters produced by task clustering represents the optimal or near-optimal number of processors on which to schedule the task graph.) For example, we experimented on two DAGs $G1$ and $G2$. DAG $G1$ has 947 nodes and DAG $G2$ has 577 nodes. Using the one-phase method, we ran LB, CTM, and RAND on DAG $G1$ assuming 128 physical processors and on DAG $G2$ assuming 32 physical processors. The makespans produced by the algorithms are given by columns 4 to 6 of Table 6.2.

Column 7 of the table gives the makespan produced by CASS-II, and the number in parentheses gives the corresponding number of clusters. Note that since for each DAG the number of clusters produced by CASS-II is already less than the number of physical processors, the cluster merging step will not be performed by

the two-phase method. Thus, for example, for DAG $G1$ the two-phase method will output a schedule using 2 processors and with makespan= 6283. On the other hand, the one-phase method using LB will output a schedule using 128 processors and with makespan= 45,556, which is 7 times worse than the two-phase method. The same is true for CTM and RAND.

	Node #	PE #	MAKESPAN			
			LB	CTM	RAND	CASS-II
G1	947	128	45556	13038	32699	6283(2)
G2	577	32	21185	7952	13118	4125(3)

Table 6.2 Experimental results for sample DAGs G1 and G2.

Grain Size	Two-Phase				One-Phase			
	Best	Next	Worst	M(W,B)	Best	Next	Worst	M(W,B)
0.1	LB	CTM	RAND	1.04	CTM	LB	RAND	1.63
0.2	LB	CTM	RAND	1.02	CTM	LB	RAND	1.18
0.3	CTM	RAND	LB	1.08	CTM	LB	RAND	1.04
0.4	CTM	RAND	LB	1.07	CTM	RAND	LB	1.19
0.5	LB	RAND	CTM	1.01	CTM	RAND	LB	1.22
0.6	LB	RAND	CTM	1.06	CTM	RAND	LB	1.33
0.7	LB	RAND	CTM	1.14	CTM	LB	RAND	1.09
0.8	LB	CTM	RAND	1.19	LB	RAND	CTM	1.07
0.9	LB	CTM	RAND	1.29	LB	RAND	CTM	1.23
1.0	LB	CTM	RAND	1.36	LB	RAND	CTM	1.20
2.0	LB	CTM	RAND	1.02	RAND	LB	CTM	1.12
3.0	LB	CTM	RAND	1.08	RAND	LB	CTM	1.08
4.0	LB	CTM	RAND	1.11	LB	RAND	CTM	1.10

Table 6.3 Relative performance of cluster merging algorithms.

We next analyze which cluster merging algorithm is superior for the one-phase and two-phase methods. Table 6.3 ranks the three cluster merging algorithms according to the makespans they generated (Best = smallest makespan). The column labeled M(W,B) gives the average makespan ratio of the worst algorithm over the best algorithm. For the one-phase method, CTM is the clear choice for fine grain task graphs, outperforming the two other algorithms for grain size 0.7 or less. On the

other hand, for grain sizes greater than 0.7, LB is generally the better method. This is to be expected because inter-task communication times dominate task execution times for fine grain task graphs (hence, minimizing communication traffic) while task execution times dominate inter-task communication times for coarse grain task graphs (hence, balance processor workload).

For the two-phase method, a different situation arises: LB is generally superior to the other two algorithms regardless of the grain size of the original task graph. The reason is that the task clustering step increases the granularity of a fine grain task graph to the point where task execution time is more or less equal to inter-task communication time. That is, the fine grain task graph becomes coarse grain. Consequently, LB is the algorithm of choice for cluster merging because the input task graph is now coarse grain. Finally, RAND exhibits somewhat erratic behavior and is generally the worst algorithm for either the one-phase or two-phase method.

Finally, Table 6.4 gives the runtimes of the cluster merging algorithms. As expected, CTM is the slowest algorithm because it computes the communication traffic between every pair of clusters. LB and RAND are significantly faster. For the one-phase method, RAND is faster than LB because LB spends extra time computing the workload of each cluster. Interestingly, for two-phase method, LB is very competitive to RAND, and in fact is faster than RAND in a majority of the cases. Our explanation for this is as follows: After the task clustering step, the clusters have granularity close to 1. As the clusters are merged, LB will always merge clusters with the smallest workload, which will also contain the fewest number of tasks. The actual merging step — which takes time proportional to the number of tasks — will thus be computed fairly quickly. On the other hand, RAND will, with high probability, merge clusters with more tasks. Consequently, the actual merging step takes longer. Thus, although RAND does not spend extra time computing the

clusters' workload (as does LB), it takes considerably more time actually merging the clusters.

Grain Size	Two-Phase			One-Phase		
	LB	CTM	RAND	LB	CTM	RAND
0.1	16	16	16	416	215491	83
0.2	16	30	33	699	236690	249
0.3	18	121	33	799	361485	183
0.4	23	584	58	766	354369	150
0.5	24	1218	33	308	95938	75
0.6	41	2564	43	433	178393	100
0.7	48	3566	58	841	447282	133
0.8	39	2561	29	250	66172	49
0.9	26	1273	66	624	269989	125
1.0	29	1719	55	641	268371	108
2.0	39	986	49	416	128278	99
3.0	44	1579	25	399	128728	116
4.0	39	1651	41	566	196942	149

Table 6.4 Runtime of cluster merging algorithms run on a Sun Sparc workstation (in msec).

6.4 Summary

The experimental results clearly demonstrate the effectiveness of the two-phase method of CASS, in which task clustering is performed prior to the actual scheduling process. Task clustering determines the optimal or near-optimal number of processors on which to schedule the task graph. In other words, there is never a need to use more processors (even though they are available) than the number of clusters produced by the task clustering algorithm — doing so would only increase the parallel execution time.

The experimental results also indicate that when task clustering is performed prior to scheduling, load balancing (LB) is the preferred approach for cluster merging. LB is fast, easy to implement, and produces significantly better final schedules than communication traffic minimizing (CTM). While CTM outperforms LB for fine grain

task graphs, such a situation never arises in the two-phase method of CASS because the task clustering phase produces coarse grain task graphs, for which LB is clearly superior to CTM.

In summary, the two-phase method consisting of task clustering and load balancing is a simple yet highly effective strategy for scheduling task graphs on distributed memory parallel architectures.

CHAPTER 7

CLUSTERING DYNAMIC TASK GRAPHS

7.1 Online Scheduling of Dynamic Trees

In this chapter, we investigate online scheduling algorithms for *dynamic trees*. Dynamic trees arise naturally in a number of important applications, e.g., divide-and-conquer, backtracking, branch-and-bound, and adaptive multi-grid algorithms. A dynamic tree consists of a finite number of nodes (or tasks), but its size is not known a priori to the scheduling algorithm. Initially, only the root of the tree is "known" and can be scheduled for execution. A known node must be executed to completion before it spawns its children (i.e., before its children become "known").

We assume that the dynamic tree is to be executed on a distributed memory parallel machine with an unbounded number of processors. Every node of the tree represents a sequential task that takes μ time units to execute on any processor of the parallel machine. In addition, if v is a child of w in the tree, then a communication delay of λ time units is incurred if v is executed on a processor different from the one that executed w. This implies that if w finishes at time t on processor p, then v can be started on processor $q \neq p$ no earlier than time $t + \lambda$. On the other hand, if $q = p$ then v can be started at time t, i.e., as soon as w finishes. The communication delay λ models the cost of migrating v to another processor, along with any output data produced by w that will be need for v's execution.

The *granularity* of a tree is an important parameter which we take into account when analyzing the performance of online tree scheduling algorithms. For a tree T with uniform node execution times μ and uniform communication delays λ, its granularity is defined as $g(T) = \mu/\lambda$. T is called *fine-grain* if $g(T) < 1$; otherwise, T is *coarse-grain*.

7.2 Competitive Analysis

We study the performance of online scheduling algorithms for dynamic trees using the *competitive analysis* approach first introduced by Sleator and Tarjan [45]. Let A be a deterministic online scheduling algorithm and let $M_A(T)$ be the makespan of A's schedule on tree T. Let $M_{OPT}(T)$ be the makespan of the schedule produced by an optimal *offline* scheduling algorithm that is given the entire tree T in advance. Algorithm A is said to be *c-competitive* (or has *competitive ratio* c) if $M_A(T) \leq cM_{OPT}(T) + O(1)$ for all trees T. If A is a randomized algorithm, then A is said to be *c-competitive* if $E[M_A(T)] \leq cM_{OPT}(T) + O(1)$ for all trees T, where the expectation is taken over all random choices of the algorithm A.

As with other online algorithms, online tree scheduling can be viewed as a game against an adversary who is allowed to determine the *requests* (i.e., tasks) that must be *served* (i.e., executed) before it issues *new requests* (i.e., children of a completed task). Thus, lower bound arguments can be phrased in terms of a strategy for the adversary that forces the competitive ratio to be as large as possible. For the deterministic case, one may assume that the adversary has complete knowledge of the online algorithm. For the randomized case, we distinguish between two types of adversaries. An *adaptive adversary* is one who knows in advance both the online algorithm and the results of the coin tosses of the algorithm. An *oblivious adversary* knows only the algorithm but not the results of the coin tosses. Our lower bounds assume an oblivious adversary, which is weaker than an adaptive adversary. Of course, using a weaker adversary means our lower bound results stronger.

7.3 Summary of Results

In [7], it was shown that finding an optimal *offline* schedule for directed acyclic task graphs is an NP-hard problem, even when restricted to trees. On the other hand, we

have shown in Chapter 3 that there is an *offline* tree scheduling algorithm which, for a tree with arbitrary granularity, produces a schedule whose makespan is at most twice optimal. It is therefore interesting to ask whether there is an *online* tree scheduling algorithm that is *c-competitive* for some constant c.

We answer this question in the negative. We show that any online tree scheduling algorithm, even a randomized one, has competitive ratio $\Omega((\frac{1}{g})/log_d(\frac{1}{g}))$ for trees with granularity at most $g < 1$ and degree d. Moreover, if the tree is allowed to have unbounded degree, the competitive ratio is $\Omega(\frac{1}{g})$. Thus, the competitive ratio grows inversely with the granularity, and implies that very bad schedules can result from online scheduling of fine-grain dynamic trees.

We also prove a tight upper bound by exhibiting a simple *deterministic* online tree scheduling algorithm that achieves a competitive ratio of $O((\frac{1}{g})/log_d(\frac{1}{g}))$. Thus, randomization *does not help* in online tree scheduling. This result is interesting in light of the fact that randomization helps in other online settings, e.g., online embedding of dynamic trees in fixed connection networks [4, 32].

7.4 The Lower Bound

Theorem 5 *The competitive ratio of any randomized online tree scheduling algorithm, working against an oblivious adversary, is* $\Omega((\frac{1}{g})/log_d(\frac{1}{g}))$ *for trees with granularity at most* $g < 1$ *and degree* d, *for any* $2 \leq d \leq \lceil \frac{1}{g} \rceil$.

Proof. Let $k = \lceil \frac{1}{g} \rceil$. The adversary's strategy is to construct a tree T of the form depicted in Figure 7.1.

T is composed of $L + 1$ identical trees $T_0, T_1, \ldots, T_L$, such that the root x_i of T_i is a leaf of the previous tree T_{i-1}, for $1 \leq i \leq L$. Each of the T_i's is a complete d-ary tree of height $h = \lceil log_d k \rceil$. Figure 7.2 illustrates the tree T for the case $d = 3$ and $k = 8$. For this case, each T_i is a complete 3-ary tree of height $h = \lceil log_3 8 \rceil = 2$.

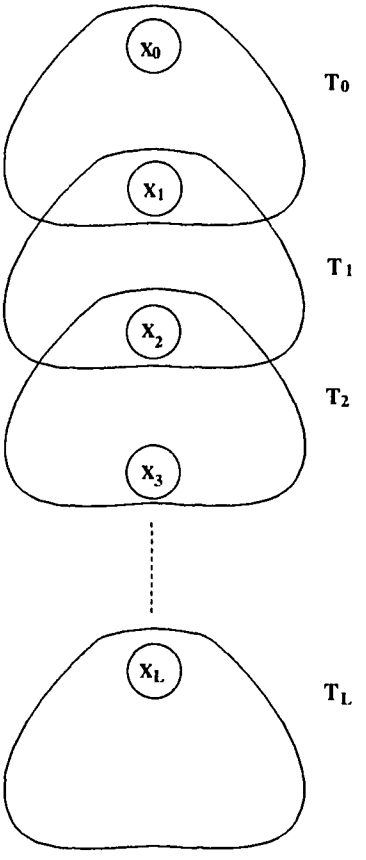


Figure 7.1 Tree T constructed by the adversary.

All nodes in T have unit execution time, and all edges have communication delay k. Thus, T has granularity $\frac{1}{k} = \frac{1}{\lceil 1/g \rceil} \leq g$. For $1 \leq i \leq L$, x_i is chosen by the adversary randomly and uniformly from among the leaves of T_{i+1}. Finally, the adversary chooses $L \geq 3k + 1$.

Let A be any randomized algorithm that schedules T online. Suppose that A executes x_i on some processor p, finishing at time $f(x_i)$. As A proceeds, it will eventually execute some number m of the d^h leaves of T_i on processor p and the rest on some other processors (m may be 0). Let $c_1, c_2, \ldots, c_m$ be the m leaves executed on p, ordered by increasing finish times: $f(c_1) < f(c_2) < \ldots < f(c_m)$. Since x_{i+1} is chosen by the adversary randomly and uniformly from among the d^h leaves, the probability that $x_{i+1} = c_j$ is $\frac{1}{d^h}$, regardless of the probability distribution used by

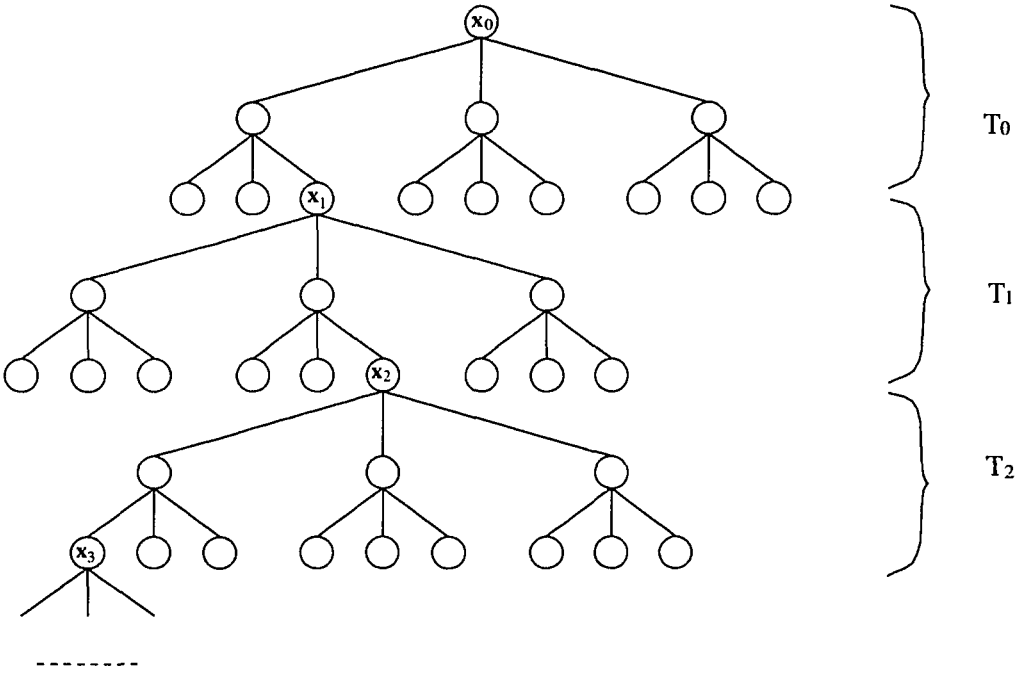


Figure 7.2 Tree T for the case d=3 and k=8.

A. Similarly, the probability that x_{i+1} is not any one of the $c_j's$ (i.e., executed on some processors other than p) is $\frac{d^h-m}{d^h}$. In the latter case, x_{i+1} cannot finish earlier than $f(x_i)+k+h$ because there are h unit-time tasks along the path from x_i to x_{i+1} (excluding x_i but including x_{i+1}) and a communication delay of at least k is incurred along the path (since x_{i+1} is executed on a processor other than p). Therefore, we have

$$E[f(x_{i+1})] \geq \frac{1}{d^h}\sum_{j=1}^{m} f(c_j) + \frac{d^h-m}{d^h}[f(x_i)+k+h] \qquad (7.1)$$

Let $f(c_j) = f(x_i)+\Delta_j$. Note that $\Delta_j \geq h$ since there are h unit-time tasks along the path from x_i to c_j (excluding x_i but including c_j). Moreover, since $f(c_j) < f(c_{j+1})$ then $\Delta_j < \Delta_{j+1}$. Substituting $f(x_i)+\Delta_j$ for $f(c_j)$ in equation (7.1) yields:

$$E[f(x_{i+1})] \geq f(x_i) + \frac{1}{d^h}\sum_{j=1}^{m}\Delta_j + \frac{d^h - m}{d^h}(k+h) \tag{7.2}$$

To find the minimum, first fix m. Since $h \leq \Delta_1 < \Delta_2 < \ldots < \Delta_m$, the minimum occurs when $\Delta_j = (h-1) + j$. Therefore,

$$E[f(x_{i+1})] \geq f(x_i) + \frac{1}{d^h}[m(h-1) + \frac{m(m+1)}{2}] + \frac{d^h - m}{d^h}(k+h) \tag{7.3}$$

When m is allowed to vary, the minimum occurs when $m = \frac{2k+1}{2}$. From this, we get

$$E[f(x_{i+1})] \geq f(x_i) + k + h - \frac{1}{d^h}[\frac{k^2}{2} + \frac{k}{2} + \frac{1}{8}] \tag{7.4}$$

But $h = \lceil log_d k \rceil$ and hence $d^h \geq k$. Therefore,

$$E[f(x_{i+1})] \geq f(x_i) + k + h - \frac{1}{k}[\frac{k^2}{2} + \frac{k}{2} + \frac{1}{8}] \tag{7.5}$$

$$E[f(x_{i+1})] \geq f(x_i) + \frac{k}{2} + h - (\frac{1}{2} + \frac{1}{8k}) \tag{7.6}$$

$$E[f(x_{i+1})] \geq f(x_i) + \frac{k}{2} + h - 1, since\ k > 1. \tag{7.7}$$

Since $f(x_0) = 1$ it follows that

$$E[f(x_L)] \geq 1 + \frac{Lk}{2} + L(h-1) \tag{7.8}$$

Now x_L is itself the root of a complete d-ary tree T_L of height $h = \lceil log_d k \rceil$. Hence

the last node to execute in T_L finishes no earlier than time $f(x_L) + h$. We therefore conclude that the expected makespan of A on T is

$$E[M_A(T)] \geq 1 + \frac{Lk}{2} + L(h-1) + h \tag{7.9}$$

We now give an upper bound on the makespan of an optimal offline schedule. Consider the schedules depicted in Figure 7.3. This schedule picks any leaf z in T_L and executes all the nodes along the path from x_0 to z on the same processor p. Let y be a node not on this path but whose parent is on the path. Node y is executed on a separate processor, together with all the nodes in the subtree rooted at y.

Figure 7.3 shows the nodes $y_0, y_1, \ldots, y_{h-1}$ not on the path from x_L to z but whose parents are along this path. Let q_i be the processor that executes the subtree rooted at y_i and let $f(q_i)$ be the finish time of the last node executed in q_i. Clearly the makespan of schedule S is $\max\{f(q_i)\}$. Moreover, among the q_i's, q_{h-1} has the maximum finish time. To see this, note the q_i executes a complete d-ary tree of height i, which has $\frac{d^{i+1}-1}{d-1}$ nodes. It follows that for $i > 0$,

$$\begin{aligned} f(q_i) - f(q_{i-1}) &= \tfrac{d^{i+1}-1}{d-1} - (\tfrac{d^i-1}{d-1} + 1) \\ &= (d^i - 1)(d-1) \\ &\geq 1, \text{ since } d \geq 2. \end{aligned}$$

Now x_L finishes at time $Lh+1$, since there are that many nodes along the path from x_0 to x_L. Therefore, the finish time of processor q_{h-1}, and hence the makespan of the schedule S is

$$\begin{aligned} Lh + 1 + k + \tfrac{d^h-1}{d-1} &\leq Lh + 1 + k + \tfrac{dk-1}{d-1}, \text{ since } d^h \leq dk. \\ &= Lh + 2k + 1 + \tfrac{k-1}{d-1} \\ &\leq Lh + 3k + 1, \text{ since } d \geq 2. \end{aligned}$$

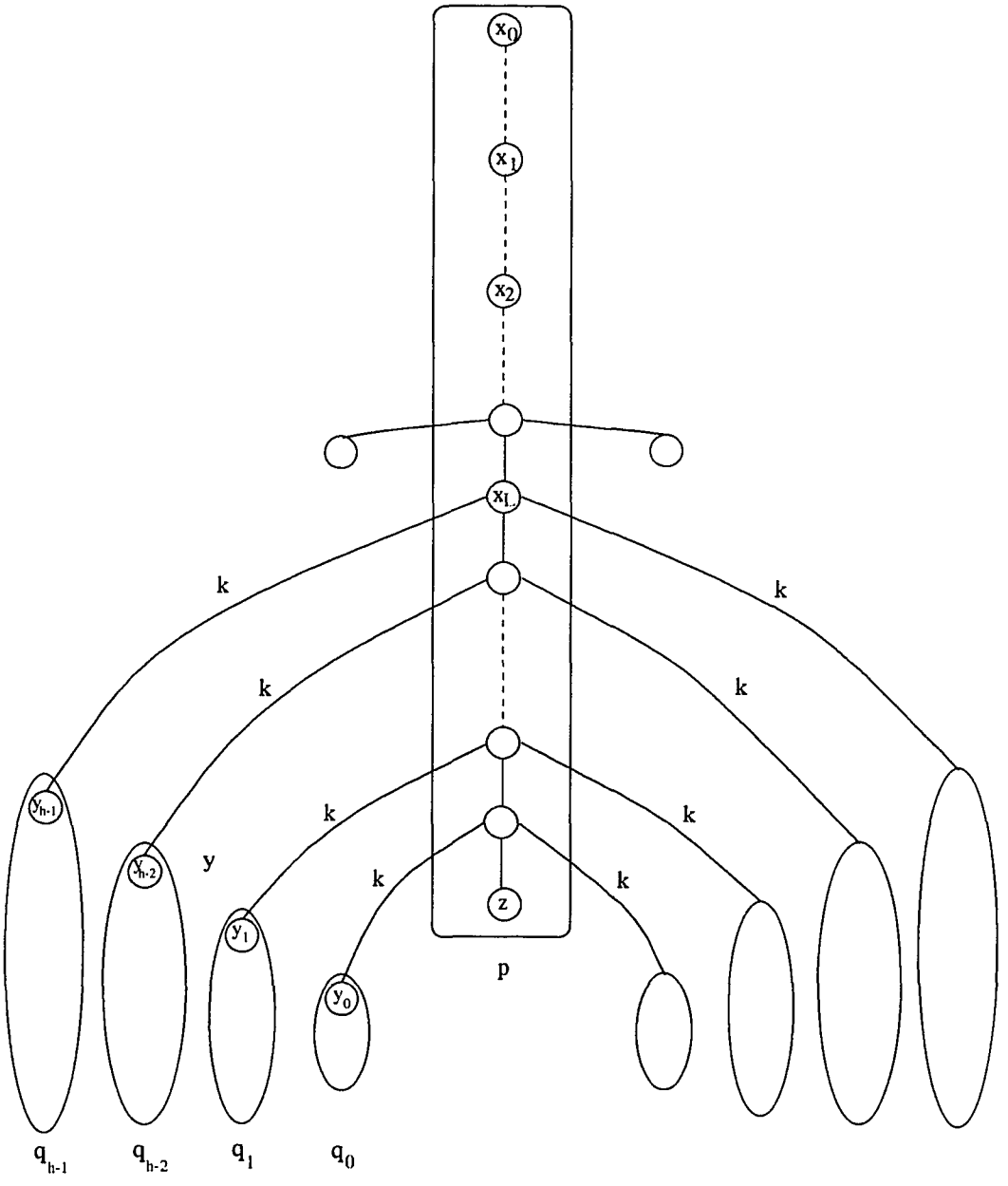


Figure 7.3 An example schedule.

An optimal offline schedule for T is no worse than schedule S; therefore,

$$M_{OPT}(T) \leq Lh + 3k + 1 \tag{7.10}$$

From Equations (7.9) and (7.10), it follows that the competitive ratio of algorithm A is

$$\begin{aligned} \frac{E[M_A(T)]}{M_{OPT}(T)} &\geq \frac{1+\frac{Lk}{2}+L(h-1)+h}{Lh+3k+1} \\ &= \frac{\frac{k}{2}+(h-1)+\frac{h+1}{L}}{h+\frac{3k+1}{L}}. \end{aligned}$$

For $L \geq 3k+1$, we get

$$\begin{aligned} \frac{E[M_A(T)]}{M_{OPT}(T)} &\geq \frac{\frac{k}{2}+(h-1)+\frac{h+1}{L}}{h+1} \\ &\geq \frac{k}{2(h+1)} \\ &= \frac{k}{2(\lceil log_d k \rceil+1)} \\ &= \Omega((\tfrac{1}{g})/log_d(\tfrac{1}{g})). \end{aligned}$$

If the tree is allowed to have unbounded degree, then the adversary can choose $d = \lceil \frac{1}{g} \rceil$. Thus, we get the following corollary:

Corollary 4 *The competitive ratio of any randomized online tree scheduling algorithm, working against an oblivious adversary, is* $\Omega(\frac{1}{g})$ *for trees with unbounded degree and granularity* $g < 1$.

7.5 A Deterministic Algorithm

In this section we present a simple deterministic online tree scheduling algorithm whose competitive ratio matches the lower bound given by Theorem 5. The algorithm is based on *bounded breadth-first clustering*. Let T be a dynamic tree with uniform node execution times μ and uniform communication delays $\lambda > \mu$. Let $k = \lceil \frac{\lambda}{\mu} \rceil =$

$\lceil \frac{1}{g(T)} \rceil$. As before, we assume that T has degree at most $d \leq k$. The algorithm processes the nodes of T in breadth-first order and groups them into clusters, such that the nodes in the same cluster are executed on the same processor. Each cluster contains the first k nodes found in a dynamic breadth-first expansion of the root node in the cluster. Every node which is a child of some fringe nodes in the cluster is assigned to a new cluster.

Formally, the algorithm proceeds in the following manner with any cluster from the time that the root node in the cluster starts its computation. We associate two entities with the cluster: a FIFO queue, Q, which keeps the currently pending nodes for that cluster, and a variable, COUNT, which maintains the current count of the nodes in the cluster that have already been executed. Initially, the queue Q contains the root node of the cluster and COUNT is set to zero. The following steps are repeated until the queue Q becomes empty.

1. Let $x = DELETE(Q)$ be the first node in the queue. Execute x on the processor associated with the cluster and increment COUNT by 1.

2. Let $y_1, y_2, \ldots, y_m$ be the children spawned by x. Let j be the largest integer $\leq m$ such that $COUNT + j \leq k$. Then add nodes $y_1, y_2, \ldots, y_j$ to the end of the queue Q and assign each of the remaining nodes (if any) to a new cluster.

We now show that the above online scheduling algorithm has optimal competitive ratio. Consider the first tree T and the associated clustering of nodes of T as produced by the online algorithm. Since the degree of T is at most $d \leq k$, any root-to-leaf path in T containing L nodes can pass through at most $N_L = \lceil \frac{L}{log_d k} \rceil$ clusters. To see this, let $C_1, C_2, \cdots, C_r$ be the sequence of clusters that are intersected by the path starting from the root of the tree. Then every cluster, except possibly C_r, is full, i.e., contains exactly k nodes. Moreover, because the nodes are processed in breadth-first order, any subpath that starts from the root of a full

cluster to a fringe node in the same cluster has at least $\lfloor log_d k \rfloor$ nodes. The bound N_L is therefore implied.

From the preceding argument, it follows that any path of length L is executed in time at most $k\mu N_L + \lambda(N_L - 1)$: the two terms respectively bound the time spent in computation and in inter-cluster communication. But $k = \lceil \lambda/\mu \rceil$ and hence $k\mu N_L + \lambda(N_L - 1) \leq \lambda N_L + \lambda(N_L - 1) < 2\lambda L$. No optimal clustering can finish the path in less than μL steps. Consequently, the competitiveness of the deterministic algorithm is at most $\frac{2\lambda N_L}{\mu L} = O((\frac{1}{g})/\log_d(\frac{1}{g}))$.

Theorem 6 *The bounded breadth-first online scheduling algorithm achieves a competitive ratio,* $O((\frac{1}{g})/\log_d(\frac{1}{g}))$, *for trees with granularity* $g < 1$ *and degree* $d < \lceil \frac{1}{g} \rceil$.

7.6 Coarse-Grain Trees

We now consider a class of out-trees where the computation time in a node dominates both the communication times from the parent node as well as the child nodes (whenever the node is scheduled on a different processor). Such out-trees, called *coarse-grain* trees, arise very often as online execution traces of recursive algorithms. Sometimes, the algorithm allows us to predict the computation times of the children of a currently executing node, along with the communication times on the edges that are grown online as a result of further recursive calls. We first consider the simple case of a coarse-grain out-tree and show that a very simple online algorithm is able to construct clusters and their associated schedules which at most 2 times the length of the optimal makespan.

Formally, let T be a tree that is being grown online and let u be any node in the tree. Then, for all pairs of nodes v, w (not necessarily distinct) which are children

of u in the tree, the condition

$$\min\{\mu(u),\ \mu(v)\} \geq \lambda(u, w)$$

defines the coarse-grain nature of the tree T. Intuitively, the communication cost on an edge (u, v) is always dominated by the computation costs at each of the endpoints of the edge as well as the computation cost of any sibling of v in the tree.

It has been shown that if a coarse-grain out-tree is known offline, then it is possible to compute an *optimal* schedule for the tree without duplication of nodes in different clusters. In fact, the algorithm produces a clustering in which every cluster is *linear*, i.e. the nodes mapped to any cluster form a contiguous path ending at some leaf node of the tree. This observation motivates a simple, greedy algorithm for constructing online clusters for the tree as it grows over time.

The strategy adopted is as follows: when a node completes execution in some processor (cluster), *any* one of the spawned children is selected and retained within the cluster for immediate execution. The remaining children are sent to new processors where they begin their own clusters. Thus, over time, the algorithm produces linear clusters; at any fixed instant, a cluster either ends in a node which is being computed, or ends in a node which will not spawn further children.

Theorem 7 *The greedy online algorithm produces a 2-optimal schedule for any coarse-grain tree T.*

Proof: Let C denote the final clustering produced by the online algorithm. From the preceding remarks, the clustering is linear: any given cluster of nodes forms a contiguous path from some internal nodes of T to some leaf nodes of T.

Fix a leaf node l in the tree and consider the unique path from the root of T to l. Let M_l be the sum of the computation costs for nodes along the path; it follows that the optimal schedule for T cannot complete processing l before time M_l. The path starts at the root and follows some sequence of nodes in a C-cluster (say c_1),

then switches to the start of another C-cluster (say c_2), follows it for a while before switching to a third cluster and so on. In general, it is easy to see that l can finish computing in time equals to M_l plus the sum of the communication costs incurred while switching from one cluster to the next along the path. The latter cost is at most M_l from the coarse-grain condition, since the cost of every edge (u, v) that switches clusters, is dominated by the computation cost of node u, and the cost of u appears exactly once in M_l.

It follows that every leaf node in T completes computation within twice the optimal completion time. Consequently, the makespan of the online schedule is within a factor of two of the optimal makespan. ■

7.7 Summary

In this chapter, we showed that any online tree scheduling algorithm, even a randomized one, has competitive ratio $\Omega((\frac{1}{g})/log_d(\frac{1}{g}))$ for trees with granularity at most $g < 1$ and degree d. Moreover, if the tree is allowed to have unbounded degree, the competitive ratio is $\Omega(\frac{1}{g})$. Thus, the competitive ratio grows inversely with the granularity, and implies that very bad schedules can result from online scheduling of fine-grain dynamic trees.

We also proved a tight upper bound by exhibiting a simple *deterministic* online tree scheduling algorithm that achieves a competitive ratio of $O((\frac{1}{g})/log_d(\frac{1}{g}))$. Thus, randomization *does help* in online tree scheduling. This result is interesting in light of the fact that randomization helps in other online settings, e.g., online embedding of dynamic trees in fixed connection networks [4, 32].

CHAPTER 8

CONCLUSIONS

The scheduling problem is a distinguishing feature of parallel versus serial programming. Informally, the scheduling problem arises because the concurrent parts of a parallel program must be arranged in time and space so that the overall execution time of the parallel program is minimized. This is a well-known NP-complete problem. The effectiveness of the heuristics depends on a number of factors – grain size, interconnection topology, communication bandwidth, and program structure.

As a result, research performed prior to this thesis inspected the possibility of finding heuristics for approximating an optimum performance. A number of heuristics have been proposed, each of which may work under different circumstances. This variation has led to confusion and misunderstanding of the heuristics.

This research investigates those most widely accepted models in static task graph scheduling and explores the dynamic scheduling for distributed memory architectures that few had touched. As a consequence of the research, we have made several general contributions:

- A clustering algorithm with task duplication (CASS-I) that produces a schedule whose makespan is at most twice optimal. Our theoretical and experimental results show that CASS-I outperforms the other existing algorithms in this problem in terms of both speed and makespan.

- A very fast algorithm (CASS-II) that has also good empirical performance for task clustering without task duplication. It has been shown in this research that CASS-II is superior to the other existing algorithms in this problem in terms of speed. In terms of makespan, CASS-II produces the best solution quality when the granularity of the task graph is less than 0.6, and has a performance

comparable to the one of DSC which is empirically the best known algorithm in this problem for the other cases.

- A better understanding of randomization applied to dynamic tree scheduling problem. We have shown that any online scheduling algorithm, deterministic or randomized, has a lower bound on the competitive ratio. We have also developed a deterministic algorithm that matches the lower bound. Thus, randomization does not help in online scheduling of dynamic trees.

- A better understanding of two-phase and one-phase methods used to solve the problem of scheduling parallel programs for execution on distributed memory architectures. We have run cluster merging algorithms solely or incorporated with CASS clustering algorithm and found that in all cases the schedules generated by the two-phase method is better than the one produced by the one-phase method. We have also shown that use the number of physical processors in a specific target machine as a parameter may result in a schedule far away from the optimal one, and a slow speed.

- A Clustering and Scheduling System (CASS) that can be integrated with existing or future compilers for parallel machines to provide facilities for automatic granularity optimization and task scheduling. Given a parallel program and a target parallel machine, a profiler generates a task graph specifying the dependencies among the tasks of the program, the task execution times and the inter-task communication delays. The task graph is used by the clustering module in CASS. The output of the scheduling module in CASS is a clustered task graph whose number of clusters matches the number of processors of target parallel machine. The output is then used by the code generator to generate machine instructions and to insert communication and synchronization primitives at appropriate points in the generated code.

Although we have contributions in static and dynamic task clustering and scheduling for distributed memory architectures, there are several directions for future work.

- After we developed CASS, the implementation of CASS incorporated with current parallel compilers will be the most important work in the near future.

- In this research, we did not address the problem of finding the task graph (i.e., program profiler). Automatic derivation of DAG parallelism is important for CASS. The general dependence analysis is NP-hard but for some special cases there exist polynomial algorithms [39, 48]. A polynomial algorithm for general cases may be developed incorporated with an interactive system to allow users to derive DAGs.

- CASS adopts simple heuristics for the processor assignment and local scheduling. Since there are no provably good and fast algorithms for these two problems, heuristic algorithms may be developed for a better performance of entire clustering and scheduling.

- For many computing applications such as particle simulations, it is often the case where it is impossible to determine the execution time of tasks, direction of branches, or number of iterations in a loop [46]. This type of problem can not be solved by static task clustering algorithms. In this research, we have derived a lower bound for any online algorithms of dynamic trees and developed a deterministic algorithm that matches this bound. However, there is no provably good online algorithm for general task graph. Heuristic algorithms may be developed for general task graphs that can be applied to this problem.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, New York: Addison-Wesley, 1974.

2. J. Baxter and J. Patel, "The last algorithm: A heuristic-basd static allocation algorithm," *Proc. 1989 Int. Conf. on Parallel Processing*, vol. 2, pp. 217–222, 1989.

3. S. Ben-David and A. Borodin. etc, "On the power of randomization in online algorithm," *Comm. ACM*, pp. 379–386, 1990.

4. S. Bhatt, D. Greenberg, T. Leighton, and P. Liu, "Tight bounds for on-line tree embeddings," in *Proc. 2nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 344–350, 1991.

5. F. Bodin and F. Charot, "Loop optimization for horizontal microcoded machines," in *Proc. Int. Conf. on Supercomputing*, pp. 164–176, 1990.

6. P. Chretienne, "Task scheduling over distributed memory machines," Tech. Report M.A.S.I. 253, Universite Pierre et Marie Curie, Laboratorie MASI, UA818, 4, place Jussie, 75252 Paris cedex 05, France., 1988.

7. P. Chretienne, "Complexity of tree scheduling with interprocessor communication delays," Tech. Report M.A.S.I. 90.5, Universite Pierre et Marie Curie, 1990.

8. Y.-C. Chung and S. Ranka, "Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors," in *Proc. Supercomputeing '92*, pp. 512–521, 1992.

9. E. G. Coffman, Jr. (eds.) *Computer and Job-Shop Scheduling Theory*, New York: John Wiley, 1976.

10. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, Cambridge, MA: The MIT Press, 1990.

11. W. Dally, "Network and processor architecture for message-driven computer," *in VLSI and Parallel Computation* (R. Suaya and G. B. eds.), San Mateo, CA: Morgan Kaufmann, pp. 140–218, 1990.

12. K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Computer*, vol. 15:6, pp. 50–56, 1982.

13. H. El-Rewini, T. G. Lewis, and H. H. Ali, *Task Scheduling in Parallel and Distributed Systems*, Englewood Cliffs, NJ: Prentice Hall, 1994.

14. G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*, Englewood Cliffs, NJ: Prentice Hall, vol. 1, 1988.

15. G. C. Fox, R. D. Williams, and P. C. Messina., *Parallel Computing Works*, San Mateo, CA: Morgan Kaufmann, 1994.

16. A. Gerasoulis and T. Yang, "Clustering task graphs for message passing architectures," in *Proc. Int. Conf. on Supercomputing*, pp. 447–456, 1990.

17. A. Gerasoulis and T. Yang, "Scheduling program task graphs on MIMD architectures," *Algorithm Derivation and Program Transformation* (R. Paige, J. Reif, and R. Wachter eds.), New York: Kluwer Publisher, 1992.

18. A. Gerasoulis and T. Yang, "On the granularity and clustering of directed acyclic task graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4:6, pp. 686–701, June 1993.

19. M. Girkar and C. Polychronopouls, "Partitioning programs for parallel execution," in *Proc. Int. Conf. on Supercomputing*, 1988.

20. R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Optimization and approximation in deterministic sequencing and scheduling: A survey", *Ann. Discrete Math.*, pp. 287-326, May 1979.

21. M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3:2, pp. 179–193, March 1992.

22. J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal on Computing*, vol. 18:2, pp. 244–257, April 1989.

23. K. Hwang, *Advanced Computer Architecture with Parallel Programming*, New York: McGraw-Hill, 1994.

24. S. Kim and J. C. Browne, "A general approach to mapping of parallel computation upon multiprocessor architectures," *Proc. Int. Conf. on Parallel Processing*, vol. 3, pp. 1–8, 1988.

25. S. J. Kim, "A general approach to multiprocessor scheduling," Tech. Report TR-88-04, University of Texas at Austin, Department of Computer Science, 1988.

26. B. Kruatrachue and T. Lewis, "Grain size determination for parallel processing," *IEEE Software*, pp. 23–32, Jan. 1988.

27. A. W. Kwan, L. Bic, and G. Gajski, "Improving parallel program performance using critical path analysis," *Languages and Compilers for Parallel Computing* (D. Gerlernter, A. Nicolau, and D. P. eds.), Cambridge, MA: The MIT Press, 1990.

28. Y. -K. Kwok and I. Ahmad, "Exploiting Duplication to Minimize the Execution Times of Parallel Programs on Message-Passing Systems", Proc. Symposium on Parallel and Distributed Systems, 1994.

29. E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Recent developments in deterministic sequencing and scheduling: A survey", *Deterministic and Stochastic Scheduling*, (M. H. Dempster, J. K. Lenstra, and A. H. G. Rinnooy Kan eds.), D. Reidel, The Netherlands: Dordrecht, pp. 367-374, 1982.

30. C. Y. Lee, J. J. Hwang, Y. C. Chow, and F. D. Anger, "Multiprocessor scheduling with interprocessor communication delays," *Oper. Res. Lett.*, vol. 7:3, pp. 141–147, 1988.

31. F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Array Trees Hypercubes*, San Mateo, CA: Morgan Kanfmann, 1992.

32. T. Leighton, M. Newman, A. G. Ranada, and E. Schwabe, "Dynamic tree embeddings in butterflies and hypercubes," *Proc. ACM Sympoisum on Parallel Algorithms and Architectures*, pp. 224-234, 1989.

33. C. McCreary and H. Gill, "Automatic determination of grain size for efficient parallel processing," *Comm. ACM*, pp. 1073–1078, Sep. 1989.

34. C. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle, "A comparison of heuristics for scheduling DAGs on multiprocessors," *Proc. 8th Int. Parallel Processing Symposium*, pp. 446–451, 1994.

35. M. A. Palis, J.-C. Liou, and D. S. Wei, "Task clustering and scheduling for distributed memory parallel architectures," To appear in *IEEE Transaction on Parallel and Distributed Systems.*

36. M. A. Palis, J.-C. Liou, and D. S. Wei, "A greedy task clustering heuristic that is provably good," in *Proc. Int. Symposium on Parallel Architectures, Algorithms and Networks*, pp. 398–405, 1994.

37. M. A. Palis, J.-C. Liou, S. Rajasekaran, S. Shende and D. S. Wei, "Online scheduling of dynamic trees," Submitted for publication.

38. J.-C. Liou, M. A. Palis, D. S. Wei, "Performance analysis of task clustering heuristics for scheduling static DAGs on multiprocessor," Manuscript.

39. W. Pugh, "The Omega test: a fast and practical integer programming algorithm for dependence analysis," *Proc. of Supercomputing '91*, pp. 446–451, 1994.

40. C. H. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," *SIAM Journal on Computing*, vol. 19:2, pp. 322–328, April 1990.

41. J. Ramanujam and P. Sadayappam, "Compile-time techniques for data distribution in distributed memory machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2:4, pp. 472–482, Oct 1991.

42. H. E. Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 138–153, 1990.

43. V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, Cambridge, MA: The MIT Press, 1989.

44. U. Schwiegelshohn, F. Gasperoni, and K. Ebcioglu, "On optimal parallelization of arbitrary loops," *Journal of Parallel and Distributed Computing*, vol. 11, pp. 130–134, 1991.

45. D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Comm. ACM*, vol. 28:2, pp. 202–208, 1985.

46. D. Towsley, "Allocating programs containing branches and loops within a multiple processor system," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 10, pp. 1018–1024, 1986.

47. M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2:4, pp. 452–471, Oct 1991.

48. M. Wolfe and U. Banerjee, "Data dependence and its application to parallel processing," *International Journal of Parallel Programming*, vol. 16, no. 2, pp. 137-178, 1987.

49. M. Y. Wu and D. D. Gajski, "A programming aid for hypercube architectures," *Journal of Supercomputing*, vol. 2, pp. 349–372, 1988.

50. M. Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1:3, pp. 330–343, July 1990.

51. T. Yang, " Scheduling and Code Generation for Parallel Architectures", PhD dissertation, Tech. Report DCS-TR-299, Department of Computer Science, Rutgers University, May. 1993.

52. W. H. Yu, " LU Decomposition on a Multiprocessing System with Communication Delay", PhD dissertation, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1984.

53. H. Zima, *Supercompilers for Parallel and Vector Computers*, New York: ACM Press, 1991.