Automatic Code Partitioning for Distributed Memory Multiprocessors (DMMs)

Moez Ayed

CENG 96-36

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213) 740-4484

November 1996

AUTOMATIC CODE PARTITIONING FOR DISTRIBUTED MEMORY MULTIPROCESSORS (DMMs)

by

Moez Ayed

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Computer Engineering)

November 1996

UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90007

This dissertation, written by

MOEZ AYED

under the direction of his Dissertation Committee, and approved by all its members, has been presented to and accepted by The Graduate School, in partial fulfillment of requirements for the degree of

DOCTOR OF PHILOSOPHY

[illegible]
Dean of Graduate Studies

Date May 9, 1996

DISSERTATION COMMITTEE

[illegible]
Chairperson

[illegible]

[illegible]

Dedication

I dedicate this thesis to my father, my brothers and sisters, and especially my mother.

Acknowledgments

I am grateful to all the people who directly or indirectly helped me through the years while I was doing my Ph.D. research.

First, I wish to express my sincere gratitude to my advisor Professor Jean-Luc Gaudiot for his assistance and guidance. I also thank the other members of my thesis committee: Professor Ellis Horowitz and Dr. Sandeep Gupta for taking the time to attend my defense exam and for their valuable comments.

My thanks also goes to my group members and friends for their assistance in various ways and for their comments.

I also would like to thank my Tunisian friends who gave me moral support and who stood by me while I was doing my Ph.D. research. My special thanks goes to Raouf Khelil, Anis Kallel, Sabeur Siala, Rym Ben Saiden, Mohamed and Mouna Sellami, Rym M'Hallah and many others.

My deepest thanks goes to my parents, brothers and sisters who never stopped lending me great support when I was going through difficult times and things were looking very uncertain. Especially, my appreciation goes to my mother who was there for me all the time and who supported me in many different ways during my PHD years. She has always been very concerned and I will never forget her words of encouragements. Her love and caring gave me enough strength to keep trying and to never give up until my goal was reached. She was always in great pain when I was facing obstacles and my frustrations were high. I had a great feeling of relief when I announced to her my success at defending my dissertation, which made her cheer so loud and which brought tremendous joy to her heart.

Contents

List Of Tables

List Of Figures

Abstract

Two of the main phases of compilers for Distributed Memory Multiprocessors (DMMs) are the code partitioning and scheduling phases. Several satisfactory solutions have been proposed regarding the scheduling phase. However, much more needs to be done regarding the code partitioning phase. Existing work regarding the partitioning problem either considers a specific application and find an efficient partitioning scheme for it (i.e. no automatic partitioning), or determine a general solution (automatic partitioning) that is too simple and therefore not efficient (e.g. exploits only one kind of parallelism level).

Our research deals with the code partitioning phase of the compiler. We propose a data-flow based partitioning method where all levels of parallelism are exploited. Given a Directed Acyclic Graph (DAG) representation of the program, we propose a procedure that automatically determines the granularity of parallelism by partitioning the graph into tasks to be scheduled on the DMM. The granularity of parallelism depends only on the program to be executed and on the target machine parameters. Our algorithm uses the Critical Path Length (CPL) of the task graph as the criterion to compare partitions. Ideally, we want to determine the task graph with minimal CPL among all possible task graphs. The output of our algorithm is passed on as input to the scheduling phase. Finding an optimal solution to this problem is NP-complete. Due to the high cost of graph algorithms, it is nearly impossible to find close to optimal solutions that don't have very high cost (higher order polynomial). Therefore, we propose heuristics that give good performance and that have relatively low cost.

Chapter 1

Introduction

1.1 Parallel Processing and Multiprocessors

Improvements in device technology are no longer capable of meeting the performance demands of today's applications. Modern sequential computers are approaching the fundamental physical limitation that signals cannot travel faster than the speed of light. However, current scientific problems requirements are ever-increasing.

Parallel processing research [3, 24, 25, 28, 34] offer architectural solutions to this problem. An example of such architectural solutions is multiprocessor systems, which are gaining more and more popularity in the research community and even in the industry. The hope behind designing these machines is that collective computational power of the multiple processors will enable us to solve very large problems. Because of the importance of multiprocessors, it is expected that these machines will become widely commercialized and widely used to solve complicated and computationally demanding problems.

The advances in hardware design of parallel computers have not been followed by corresponding advances in software to program these machines. This is especially true for Distributed Memory Multiprocessors (DMMs)[1], for which there is no shared memory that can be used by all the Processing Elements (PEs)[2]. High-level programming abstractions for these machines are almost non-existent,

[1]From now on, Distributed Memory Multiprocessor will be abbreviated to DMM.

[2]From now on Processing Element will be abbreviated to PE.

leaving the programmers the task of explicitly programming these architectures using machine-dependent, low-level abstractions. This approach is error-prone and forces the programmer to deal with many details outside of the application domain. More precisely, the programmer has to deal with all parallel processing tasks required to program the parallel machine. These tasks include explicit partitioning of the program code into parallel tasks, scheduling these tasks on the PEs, synchronization, and explicit distribution of data among the PEs and insertion of the appropriate message passing calls needed to exchange data from one remote memory to another.

Because of the problems mentioned above, providing solutions to ease the task of programmers of multiprocessors has become a very active area of research in the last few years [1, 4, 5, 11, 12, 26, 27, 32, 38, 39, 40, 43, 45, 55, 60]. Much effort is being done to make the parallel processing tasks mentioned above be done automatically by the compiler of the parallel machine. This way, the user does not have to know the details of the architecture of the machine. His/her main concern is the specification of the algorithm for solving the problem.

Several languages have been proposed to program multiprocessors. Some of these languages are concurrent PASCAL, ADA, OCCAM, parallel FORTRAN and parallel C. In all of these languages, the programmer has to express the parallelism explicitly. Furthermore, the programmer is responsible for partitioning the program, allocating tasks on different PEs and scheduling the execution of the tasks on the PEs. Hence, programming multiprocessors is still a very complicated problem.

In order to overcome this problem, much research is being done to efficiently program multiprocessors using functional languages. For that to be possible, we need to come up with sophisticated compilers that have powerful optimizers so that the implementation of these languages can be efficient.

The gain from that is:

- Programming using functional languages is very user friendly and is faster than using imperative languages, since reading and correcting functional programs is much easier than reading and correcting conventional languages.

- When using functional languages, the user doesn't have to know anything about the details of the machine: free the programmer from the details of the machine.
- Parallelism is expressed implicitly in the program and is extracted automatically by the compiler.
- Partitioning, allocation and scheduling are done automatically by the compiler rather than by the programmer.

All of these advantages can be summarized in saying that using functional languages to program multiprocessors makes the software much easier to produce and maintain and thus cheaper. The programmer is provided with a very high level language, where the main concern is the specification of the algorithm for solving a problem.

Furthermore, if we compile a functional programming language for a whole group of multiprocessors, then we will not need to rewrite a program executed on one of these machines, if we need to execute it on another machine in this group (portability of software).

1.2 DMMs: The computers of the future

In order to be able to solve the very large problems that face our scientific community today, we need computers capable of supporting thousands of powerful processors, whose aggregate computing capabilities are sufficiently strong. Shared memory multiprocessors cannot support a big number of PEs, and therefore cannot be used to solve this kind of problems efficiently. DMMs on the other hand are potentially scalable to a very large number of PEs, and hence are the right kind of machines to solve these large-scale problems. A major difficulty with the current generation of Distributed Memory Machines is that they generally lack programming tools for software development at a suitably high level. The user has to deal with all aspects of the distribution of data, since he is provided with separate address spaces, all aspects of the distribution of work load to the processors, must explicitly take care of the inter-PE communication by using communication constructs to send and receive data[3], and must control the program's execution at a very low level. This results in a programming style similar to assembly programming on a sequential machine. This is tedious, time consuming, and error prone. The programmer has to face several issues that do not have their counterparts in sequential programming, such as deadlock which is a major challenge for programmers of multiprocessors. The programmer also has to decide when it is advantageous to replicate data across processors, rather than send data. Moreover, debugging could be extremely difficult. This has resulted in very slow software development cycles and, in consequence, very high software costs. This research is an attempt at making programming DMMs very user friendly and therefore make the software cost be low. Our main objective is to provide the user with a machine independent programming model which is easy to use, and at the same time performs with acceptable efficiency. This will make the software portable to different DMMs. Furthermore, changing the parallel program to reflect a change in the specifications of the problem will be an easier task. Some examples of DMMs are: CM5, T3D, Intel Paragon and the Hypercube.

1.3 Outline of this research

Our research deals with the code partitioning phase of the compiler. We propose a data-flow based partitioning method where all levels of parallelism are exploited. Given a Directed Acyclic Graph (DAG)[4] representation of the program, we propose a procedure that automatically determines the granularity of parallelism by partitioning the graph into tasks to be scheduled on the DMM. The granularity of parallelism depends only on the program to be executed and on the target machine parameters. Our algorithm uses the Critical Path Length (CPL) of the task graph as the criterion to compare partitions. Ideally, we want to determine the task graph with minimal CPL among all possible task graphs. The output of our

[3]This is called message passing.

[4]From now on Directed Acyclic Graph is abbreviated to DAG.

algorithm is passed on as input to the scheduling phase. Finding an optimal solution to this problem is NP-complete. Due to the high cost of graph algorithms, it is nearly impossible to come up with close to optimal solutions that do not have very high cost (higher order polynomial). Therefore, we propose heuristics that give good performance and that have relatively low cost.

The rest of this thesis is organized as follows: chapter 2 is about the motivation of this work. In chapter 3 we define the partitioning problem. Chapter 4 describes the analysis done to determine the choice of heuristics. Chapter 5 describes the partitioning heuristics. Chapter 6 talks about the performance analysis of our algorithm. Finally, chapter 7 summarizes our work and talks about possible future research.

Chapter 2

Background Research

2.1 Programming styles for multiprocessors

Programming multiprocessors can be broadly classified into 4 methods:

1. **Explicit Imperative Programming**: In this case, we use an imperative parallel language such as parallel Fortran or parallel C to program the DMM. The user is responsible for all dependence analysis and for inserting the parallelizing and synchronizing statements in the correct place. In addition, the programmer is responsible for the data distribution and the data movement statements (for DMMs only). This programming style is comparable to assembly programming for sequential machines. It is very time consuming and error-prone yet usually produces the most efficient code.

2. **Implicit Imperative Programming**: Here the programmer uses a conventional sequential language such as Fortran, Pascal or C. It is the job of a very intelligent compiler to extract the parallelism in the program using data dependence analysis, insert the appropriate parallelization and synchronization primitives, distribute the data across the PEs and insert the required message passing routines for inter-PE communication (for DMMs only). Usually the data dependence relations for imperative languages are quite obscure and many false dependencies exist. Therefore, the compiler is forced to make very conservative decisions. This results in under-parallelization of the program. Hence, it is generally very difficult to design such compilers which are efficient for a wide range of applications.

3. **Hybrid Implicit/Explicit Imperative Programming**: In this programming style, the language used is an extension of an existing sequential imperative language such as Fortran or C. In addition to the usual code, the programmer is responsible for specifying the data layout (distribution of data across the PEs) or the processors on which different pieces of code will execute (such as different iterations of a loop), or both. These specifications could be part of the source code, or in the form of compiler directives or pragmas. With the help of the user specifications, it will be much easier for the compiler to perform the tasks required to parallelize the code. An example of such a programming language is Fortran D [26]. Also, another approach to this programming style is to use an imperative language augmented with some explicit parallel statements. In this case, the user explicitly specifies which statements or pieces of code execute in parallel. Again, this facilitates the analyses done by the compiler.

4. **Functional Programming**: The above mentioned problems with imperative languages have led to the investigation into other kinds of languages. Functional languages are an example of that. Here, the programmer uses a functional language to write the code. All the user needs to know is the programming language that he/she is using, without any concern with the details of the machine on which the program is going to execute. It is the job of the compiler to produce the target program which is executable directly on the target machine, and compiled using the local compiler. This means that the compiler is responsible for partitioning the code (i.e. creating the parallel tasks), scheduling these tasks on the PEs (i.e. task distribution), managing the tasks for efficient execution on the PEs, and memory management (e.g. distributing the data across the PEs so that the number of remote references is minimized). Experience has shown that designing such a sophisticated compiler is a very hard problem. For example, both optimal partitioning and optimal data distribution problems are NP-complete. Most existing compilers rely on programmer interventions to help the compiler

with the analyses. This is done by either enabling or forcing the programmer to give some hints to the compiler regarding data distribution, task distribution and management, or both.

In summary, there is a tradeoff of performance for programming effort. The more explicit programming DMMs is, the better the performance is but the more the programming effort becomes. The more implicit we make this task, the less the programming effort gets at the expense of lower performance. We are faced with the challenging task of providing the programmer with a high level language, capable of abstracting the underlying architecture, implicitly detecting the parallelism in the program, and managing the parallelism for efficient execution on a wide range of multiprocessor systems. This should not come at the expense of performance. This task is obviously very challenging.

Because of all the above mentioned points, we are convinced that functional languages are the right programming languages to use, in order to have good programmability for multiprocessors.

2.2 Existing Implementations of Functional Languages on Multiprocessors and their Inefficiencies

To this day, there is no satisfactory programming environment for multiprocessors, even using functional languages. This is especially true for DMMs. Most of the inefficiencies associated with the existing methods have to do with the code partitioning, data partitioning and scheduling.

2.2.1 SISAL

2.2.1.1 Overview

SISAL[1] [33] is a general purpose functional language that supports data types and operations for scientific computation. It is intended for use on a variety of sequential, vector, multiprocessor and data-flow architectures. A primary goal in the design of SISAL was to express algorithms for execution on computers capable of highly parallel operation. It is expected that SISAL will evolve into a general purpose programming language targeted to run on future parallel computers. Being an applicative language, SISAL uses *functions* for *all* operations to aid the identification of concurrency. This results in a language with very clean semantics. In addition, SISAL has an elegant functional representation in its intermediate forms (the data-flow graphs IF1 and IF2). The language syntax, being similar to Pascal, is easy to learn and read.

SISAL is a strongly typed language. All inputs and outputs of expressions and functions are *values* (no memory address references are used). Each value has an associated SISAL data type. There are *basic* scalar arithmetic types (character, boolean, integer, real and double precision) and *aggregate* types (arrays, records, unions and streams).

SISAL supports both sequential (*non-product form*) and parallel (*product form*) loop constructs. The non-product form resembles sequential iteration in conventional languages, but retain single assignment semantics. The product-form loop allows the programmer to specify iterations that do inner (dot) and outer (cartesian) array and stream index computations. All iterations should be independent of one another. The programmer uses this construct to express parallelism explicitly. In addition to iteration forms, SISAL supports program structures for conditional execution. Note that all structured expressions and functions in SISAL can produce two or more values via *multi-expression*: comma separated lists of

[1]Researchers at the Lawrence Livermore National Laboratory (LLNL) in collaboration with individuals from the University of Manchester, Colorado State University and the Digital Equipment Corporation have developed the programming language SISAL (Streams and Iteration in a Single Assignment Language).

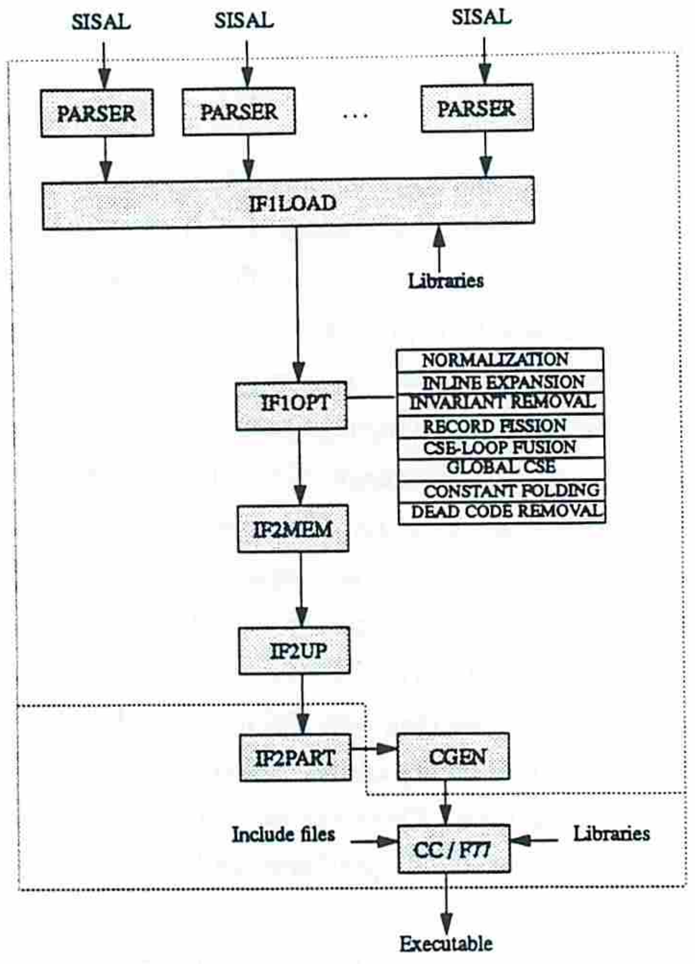


Figure 2.1: Internal Structure of OSC

expressions producing values of any type. Such expressions are both convenient and well suited for parallel evaluation.

2.2.1.2 Compilation

SISAL program → Target Machine M (a Parallel Computer, e.g. a shared or distributed memory multiprocessor).

The SISAL compiler consists of 3 parts: a front end, a back end, and a run-time system [9, 44, 8, 16, 50, 53]. Figure 2.1 shows the internal structure of an existing compiler for SISAL, developed at LLNL. This compiler is called OSC (Optimizing Sisal Compiler).

1. *Front end*: Architecture Independent.
   SISAL $\rightarrow$ IF1 graph.
   In this step, the syntax analysis of the SISAL program is done. Next, the program is translated into an intermediate dependence graph form in IF1. IF1 [51] is data-flow graph language which is applicative.

2. *Back end*:
   IF1 graph $\rightarrow$ optimized IF1 $\rightarrow$ IF2 graph $\rightarrow$ optimized IF2.
   Optimized IF1/2 $\rightarrow$ Language L directly executable on M $\rightarrow$ use M's local L compiler $\rightarrow$ Execute on M.
   Some optimization techniques are used on the IF1 graph to get an optimized IF1. Next, the IF1 graph is extended into an IF2 graph [54] which is a superset of IF1, consisting of the IF1 graph plus some memory requirements and specifications. More precisely, in the IF2 graph we attempt to preallocate array storage whenever possible, in order to reduce array copying that results from the incremental aggregate construction problem. IF2 is not an applicative language since it directly references and manipulates memory. This optimization phase from optimized IF1 to IF2 is called *build-in-place* analysis. The next phase consists of the *update-in-place* analysis. Here the IF2 graph is further optimized to help identify at compile-time those operations that can execute in-place, and to improve chances for in-place operations at run-time when the analysis fails. The result of this phase is the optimized IF2 graph. Note that both build-in-place and update-in-place analysis are optimization phases, that try to reduce the aggregate copying overhead incurred due to the single assignment nature of SISAL.

3. *Run-time system*:
   This is the library software that provides support for parallel execution, storage management and interaction with the user. This library of routines is called from the program L generated by the SISAL compiler. Then program

L is compiled using M's local L compiler, and the result is linked with the run-time system and executed on the target machine.

The compiler analysis up to the optimized IF2 graph is done by the SISAL group at LLNL. The analysis up to the optimized IF1 is completely independent from the architecture. The analysis from the optimized IF1 through the optimized IF2 is architecture independent, but was done with the assumption that the target machine has a single shared memory. All aggregate data is assumed to be allocated to a contiguous block of memory.

Portability of OSC

OSC was designed primarily to target shared memory multiprocessors. Complete implementations exist for various shared memory machines. It is quite easy to porte OSC to different shared memory multiprocessors. All what is needed is to make some minor modifications to some low-level routines and some library routines to reflect the new run-time system and low-level routines of the new target machine.

It is however much harder to porte OSC to DMMs. This is so because when writing a parallel program to target a DMM, we have to deal with the data partitioning, which is not an issue for shared memory multiprocessors. Hence the compiler has to take care of the non-local memory accesses and the message passing mechanism, which are not included in the OSC compiler. However, the parts of OSC which are architecture independent can still be used.

As for our project, we can use all the OSC analysis which is architecture independent. In addition, the analysis that includes IF2 and the corresponding optimizations can be used as well, despite the fact that IF2 was designed with a single address space in mind. This is true because as we will see later in the proposed research, the virtual shared memory mechanism is used. As for the graph partitioning, our proposed method is much more complex than the one used in OSC, and therefore that part of the compiler will have to be redesigned.

2.2.1.3 Current Implementations

SISAL has proven to be an efficient language for solving many scientific applications. It can be executed on both conventional and novel architectures. Current implementations of SISAL exist for sequential machines and for shared memory multiprocessors [9, 16, 6, 7, 52, 2, 56, 37, 35, 10, 30, 29, 36]. It has been targeted on most Unix-based uniprocessors. There is also ongoing research in distributed memory SISAL implementations [19, 20, 21, 23, 22, 41]. Also the intermediate data-flow graph representation of SISAL programs can be executed on data-flow machines.

SISAL competed very well with sequential and parallel execution performance of imperative languages such as C and Fortran, on uniprocessor machines as well as various multiprocessors and vector architectures.

However, we still have to come up with efficient implementations of SISAL on DMMs, that can compete with conventional languages implementation on these machines.

Some of the machines on which SISAL was implemented are: Sun workstations, Vax machines, Macintosh II, Sequent Balance, Cray X/MP, Alliant FX/8, Encore Multimax, Warp machine, Connection Machine, nCUBE/2, HEP, Transputers, and the University of Manchester Data-flow Machine.

2.2.1.4 Inefficiencies of OSC

The main problem with the OSC compiler is the simplicity of the partitioning scheme used. It is syntax based and exploits the parallelism used in FORALL loops only. Hence the granularity of parallelism is defined by language constructs and the programming style affects the multiprocessor performance.

2.2.2 VISA

VISA [21, 23, 22] is a system that targets SISAL to DMMs. It uses the same simple partitioning scheme used by OSC. The virtual shared memory paradigm is used for data partitioning. The mechanism for translating virtual addresses to physical ones is very costly, and therefore introduces tremendous run-time overhead. This system uses a dynamic scheduling scheme, where each PE keeps its own ready queue. This introduces much run-time overhead. Also, because of the distributed ready queues, this method causes load imbalance.

2.2.3 Occamflow

Occamflow [31, 17] is an implementation of SISAL on a distributed memory multiprocessor of transputers. The compiler takes as input a SISAL program, and generates an OCCAM program loadable on the network of transputers. Because the target code of the compiler was OCCAM, many drawbacks followed. First of all, using OCCAM the router (for communication between PEs) has to be explicitly written as part of the code. Because of the nature of the OCCAM programming environment, there is no way for the compiler to generate this router automatically, or to produce a universal router that works for all applications. Hence, the programmer has to write this router manually in OCCAM. In addition, the programmer has to add some OCCAM code to the output generated by the compiler. For example, all variable declarations in OCCAM have to be written manually by the programmer. Furthermore, since OCCAM does not allow recursive function calls, no implementation for recursive calls was done. More importantly, the partitioning scheme was too simple (syntax based) and was not even implemented. The compiler generates the code for one transputer and it is the job of the programmer to partition and load the code on the network. This was mainly due to the primitive nature of the programming environment of the transputers available at that time (for example, there wasn't any operating system available that provides routines that take care of the low level details, such as the routing between PEs). Also, the data partitioning has to be done manually by the programmer using OCCAM.

2.2.4 TAM (Threaded Abstract Machine)

2.2.4.1 Brief Description

TAM [13, 14] defines a self-scheduled machine language of parallel threads, which provides a path from data-flow program representations to conventional control flow. It presents a model that exploits fine grain parallelism and fine grain synchronization, without any specialized hardware support (with minimum hardware support). It is an attempt to prove that exploiting fine grain parallelism is a compiler and program representation issue rather than a hardware issue, and that a conventional parallel machine coupled with the right program representation and the right compiler is able to do that efficiently.

The overall goal in compiling to TAM is to produce code that is latency tolerant, yet obtain processor efficiency and locality.

All memory transactions and message passing primitives are split-phase. This encourages a latency tolerant style of code generation.

All synchronization, scheduling, and storage management is explicit and under compiler control, yet dynamic. This enables the compiler to optimize the use of processor resources for the expected case rather than the worst case.

The TAM model shows that implicit scheduling in hardware is of questionable value, as it prevents register usage beyond thread boundaries. Exposing scheduling to the compiler allows it to synthesize particular scheduling policies in specific portions of the program.

Note that the goal of TAM research is not to prove that the exploitation of fine grain parallelism is the most efficient approach, and that it is better than the other existing methods. It is merely an attempt to come up with a software approach for fine grain parallelism, and see how much performance can be obtained from it. In fact, all TAM performance results give statistics regarding context switches frequency, dynamic thread length, duration of a quanta, etc. Nothing is mentioned about absolute performance of TAM, such as total execution time of real applications, and their comparison with the currently existing approaches. It is quite obvious that the absolute performance of TAM is slower than other approaches that use a coarser grain parallelism.

2.2.4.2 Drawbacks

In attempting to exploit fine grain parallelism without any hardware support, TAM tends to introduce extra run-time overhead due to the software support approach needed to implement its model. This overhead is due in part to the explicit scheduling of threads and frame activations. Exploiting fine grain scheduling is an expensive process. Although the fine grain parallelism in TAM is exploited within a single processor and not across the processors[2], it still generates more inter-PE communication overhead than coarse grain parallelism. Because fine grain parallelism is exploited within single processors, roughly the same amount of remote data will be requested by the processors even if we had a coarse grain parallelism. However, since the code inside each processor is divided into threads and inlets, it is more likely that a larger number of smaller messages will be requested, for threads and inlets use smaller messages. Since each message sent on the network has a start-up time in addition to the time taken to communicate the data, this will result in higher communication overhead. In addition to the above problems associated with the TAM model, we have to mention that it would be difficult to come up with an automatic compiler to target a high-level language to the parallel assembly language defined by TAM to represent programs (TL0). It is quite hard to partition the code into threads and inlets that result into an efficient execution. Furthermore, code-blocks[3] correspond to function bodies and loop bodies. Hence the partitioning into code-blocks is too simple.

2.2.5 Sarkar's Work

Sarkar [47, 46, 48, 49] developed a partitioning and scheduling method for functional languages represented by data-flow like graphs. The graphs that he used are a generalization of the IF1 graph used for SISAL. His method targets both shared and distributed memory multiprocessors.

The multiprocessor model is applicable to all kinds of multiprocessors, including both shared and distributed memory multiprocessors. This model was so general that it didn't represent accurately real machines and was too simple.

[2]Code-block activations are distributed across the processors, but each code-block activation is mapped to a single processor, and therefore all threads within a code-block are executed inside a single processor.

[3]This is the unit of parallelism between processors, since each entire code-block activation is mapped to a single processor.

For instance, the architecture model does not take into account the true characteristics of DMMs and their limitations, such as the high cost of inter-processor communication.

Also, the program execution model is not efficient for DMMs. It allows any compound node in the graph to execute in parallel, in which case the compound node and nodes belonging to it (i.e. nodes that belong to subgraphs of the compound node) execute in separate processors. For DMMs this is not efficient since it could generate too much communication overhead at run-time. For instance, LOOPA and LOOPB nodes[4] are allowed to execute in parallel. To do this, the nodes that belong to the subgraphs of the LOOP[5] node are distributed across the processors, and the processor where LOOP node executes is responsible for distributing the input(s) and gathering the output(s) of the LOOP node. This generates too much traffic in the network, since for each iteration of the loop, we have to communicate messages between the processors involved in the execution of the LOOP node and nodes belonging to its subgraphs.

Another problem with Sarkar's approach (refer to Compile-time Partitioning and Scheduling part) has to do with compound non-FORALL nodes which are macro nodes[6] (let's call these nodes n_m). When partitioning a graph g, all subgraphs of the n_m nodes are partitioned first (using a recursive call to the partitioning algorithm), then the n_m nodes are assigned to tasks. Therefore an n_m node and nodes belonging to its subgraphs will belong to different tasks. Hence the tasks will not be guaranteed to be independent of each other. This violates the convexity constraint and makes the compiler analysis more complicated.

The code partitioning method proposed by Sarkar is too simple. More will be said about this later in this chapter.

Finally, Sarkar's work does not solve the data distribution problem for DMMs.

[4]LOOPA and LOOPB nodes correspond to the (Repeat ... Until) and (While ... Do) constructs respectively in SISAL.

[5]LOOP stands for LOOPA and LOOPB nodes.

[6]A macro node is a node that is allowed to execute in parallel.

2.3 Main Phases of Compilers for DMMs

In addition to all phases required by compilers for conventional sequential machines, compilers for DMMs also include phases required for parallel processing. The most important of these phases are: identification of parallelism, program code partitioning, scheduling, data partitioning (also called data distribution), and insertion of the appropriate message passing calls needed to exchange data from one remote memory to another.

2.3.1 Forms of Parallelism

The parallelism in a program is exposed implicitly by a programming language or a compiler, or explicitly by the programmer. The **granularity** of parallelism is the size of the schedulable unit of parallelism, called **grain**. The different forms of parallelism are characterized by the grain size and are as follows:

- *Procedure or loop* level parallelism uses entire loops or procedures (or functions), or different iterations of the same loop as grains. Because the granularity here is large, we call this *coarse grain* parallelism.

- *Thread* level parallelism uses basic blocks as grains. A basic block is a sequential piece of code that is of medium size, and that does not contain any loops or jump instructions. These blocks are also called *threads*. The thread is called *blocking* if it has long latency operations, such as *read* and *write*. It is called *non-blocking* if it does not have any long latency operations. This form of parallelism is called *medium grain* parallelism.

- *Instruction* level parallelism uses individual instructions as grains. Since the granularity in this case is very small, this is called *fine grain* parallelism. This offers the largest amount of parallelism at the expense of much run-time overhead.

Managing the exposed parallelism in an application is a hard problem. Once the program is partitioned into grains, we need to do the scheduling, load balancing, and synchronization among other tasks. As the granularity decreases, so does the overhead of scheduling and load balancing. However, the synchronization overhead increases as the granularity decreases [48].

2.3.2 Identification of Parallelism

During this phase, we have to identify all operations that can execute concurrently. Usually this is done by drawing the dependency graph of the program. For imperative languages, this task is quite difficult, due to the side effects caused by the updating of variables. For functional languages, this process is quite simple, since all data refers to values and not memory cells. The parallelism is implicit at all levels, and data dependencies are the only sequencing constraints. As soon as an instruction has all its data ready, we can safely execute it. Even though IF2 is not purely applicative, it is designed in such a way that data dependency ensures that when all data of any actor is ready, we can safely execute it.

2.3.3 Program Code Partitioning

Once the parallelism is identified, the first thing that we have to do in the back end compiler analysis is to partition the IF2 (or IF1, depending on which intermediate form we use) graph.

Partitioning of a parallel program is the separation of program operations into sequential tasks that can be executed concurrently. In other words, the partition specifies the sequential units of computation in the program[7]. More precisely, during this phase, we group the concurrent operations identified during the previous step into sequential tasks to be executed in parallel. Therefore, when partitioning the code, one of the things that we have to decide on is the granularity of the partition[8]. There is a trade-off between fine and coarse grain partitioning. The finer the partition is, the more available parallelism we have, and the smaller the load balancing overhead is. However, this comes at the expense of higher overhead to exploit parallelism, such as higher communication and synchronization overhead.

[7]We call these sequential units tasks.

[8]We can have fine grain partitioning, coarse grain partitioning, or something in between.

For DMMs, it is particularly important to reduce both inter-PE communication and load imbalance.

The partitioning problem for a general DAG (Directed Acyclic Graph) where nodes represent computations and arcs carry the data values is NP-complete, ruling out the possibility of a pseudo-polynomial algorithm. Therefore all we can do is to try to come up with a heuristic algorithm that gives us a performance as close to the optimal one as possible. Generally, this is still a very hard problem. An algorithm that gives a satisfactory solution has yet to be found. Some partitioning methods were proposed in [48, 47, 46, 49, 57, 12].

The partitioning of a program can be done either at compile-time or at run-time. Run-time partitioning has the advantage of using run-time information about the behavior of the program, which may lead to a better partition. However, this comes at the expense of introducing tremendous extra overhead during program execution. Hence, the partitioning algorithm has to be very simple. We can also have a hybrid compile-time/run-time partitioning. Here, an initial partition is done at compile-time. Then at run-time, we can use some information regarding the behavior of the program, to repartition the code and come up with a better partition. This method suffers from the same drawbacks associated with the pure run-time scheme. Mainly, it introduces too much run-time overhead.

The different partitioning methods for distributed memory machines can be classified as follows:

2.3.3.1 Construct based Partitioning

This is also called *syntax-based* partitioning. Here we try to exploit only the parallelism offered by the syntax of the language. For example, in Sisal we have the pipelined parallelism in streams and concurrency in FORALL loops. Function calls could also be spawned as separate tasks. Since in this case the granularity of parallelism is defined by language constructs (e.g. compound expressions and user-defined functions), the programming style dramatically affects the multiprocessor performance. Clearly, this is indesirable.

This is the simplest partitioning method in terms of analysis of the program. However it usually gives us the least amount of parallelism.

2.3.3.2 Data-flow based partitioning

In this method, we try to exploit all kinds of parallelism available in the DAG. Any two nodes are allowed to execute in parallel, provided that no data dependency exists between them. Any node is allowed to execute as soon as all input data is available.

For imperative languages, much analysis is required to determine all the parallelism available in the program. However for functional languages, this is a straight forward task.

2.3.3.3 Function-level partitioning

All functions which are independent of each other may be executed in parallel. For functional programming, two functions are independent of each other if the input of neither one is the output of the other. The *Church Rosser* property ensures that the order of evaluation of functions which are independent of one another will not affect the outcome of the program. This method might result in a granularity which is too coarse, and therefore we might not get enough parallelism to keep all PEs busy.

2.3.3.4 Data driven code partitioning

In this approach, the data is partitioned and mapped to the PEs. A processor is then thought of as owning the data assigned to it; these data elements are stored in its local memory. Then the work is distributed according to the data distribution: computations that define the data elements owned by a processor are performed by it. This is called the **owner-computes** paradigm[9]. Note that

[9]The **owner-computes** rule states that all computations updating a given datum are performed by the processor owning that datum.

we can apply the **owner-stores** paradigm[10] instead. Our hope in doing so is that the code will be mapped to the PEs in such a way that all (or at least most) of the data references are local. This results in lesser communication on distributed memory machines.

The success of this approach is very program dependent. For programs where the information about some of the data (e.g. array size) can only be determined at run time, the implementation of this method might be quite difficult and could introduce much run time overhead.

Furthermore, this approach over-emphasizes the locality issue. When we distribute the data first and then distribute the code in such a way to preserve locality of reference, the resulting code partition may be unbalanced, leading to poor performance.

For an example of a data driven code partitioning approach, refer to [26].

2.3.3.5 Code based data allocation

In the code based approach, the program is partitioned so that each processor gets approximately an equal share of the program code (load balancing). Then, depending on the data references, the data is allocated to different PEs so that communication is minimal. Here, we can also apply the owner-computes or owner-stores paradigms (or possibly a mixture of the two).

Again, the success of this method is very program dependent. For programs where the information about some of the data (e.g. array size) can only be determined at run time, the implementation of this method might be quite difficult and could introduce much run time overhead. Furthermore, the goals of locality of data and load balanced code could be conflicting, and for certain types of codes impossible to achieve simultaneously. In addition, this approach may lead to high communication overhead, that would nullify all the benefits of parallelism.

[10]The **owner-stores** rule states that the right-hand side expression of an assignment is computed by a processor which owns data appearing in that expression and this result is then sent to the processor owning the left-hand side datum.

2.3.3.6 SPMD Model of Computation

In the SPMD (Single Program Multiple Data) approach, also called Data Parallel Model of Computation, we make use of the regularity of most numerical computations. The processors execute essentially the same code in parallel, each on the data stored locally. In other words, the multiprocessor executes in a similar way to an SIMD (Single Instruction Multiple Data) machine. Note however that this approach is applicable only to specific constructs like forall loops and not all algorithms can be executed in this way. An Algorithm that can be handled using this approach has to be some code that is executed several times in parallel using multiple sets of data, such as a parallel loop. The advantages of this model of computation is its simplicity and the fact that there is no inter-PE communication. However, for most real life algorithms, there is no way of avoiding communication between PEs.

2.3.3.7 Programmer intervention

Sometimes, the user is required to supply information to the compiler, through compiler directives, assertions, etc., with regard to global, high-level properties of the algorithm whose detection by even the most able systems may be intractable. One example of this is the specification of FARALL loops to indicate the possible parallel execution of loop iterations. Another example is when the programmer asserts some information about some variable in the code (e.g. the variable is a prime number), which enables the compiler to make some decisions regarding the execution of that code.

Some parallel languages require the programmer to specify some information, regarding the partitioning of the program, in the source code. For instance Hiranandani et al. [26] use the data driven partitioning scheme. They define language extensions to Fortran called Fortran D. In this language, they include constructs for managing data distribution in non-shared address spaces. The user is responsible for specifying the data layout. Then the compiler uses the information regarding the data structures decomposition and the owner-computes rule to partition the program.

This approach makes the task of the compiler much simpler. However, the goal of making the programming of multiprocessors easy and convenient, and freeing the programmer from the details of the machine is defeated here. Furthermore, due to compile time unknowns, this method might lead to poor performance for some programs.

Many researchers believe that no compiler can on its own suffice to support the highly complex and challenging task of producing efficient programs for parallel systems [60, page 285]. They believe that advanced compilation systems will be integrated into a sophisticated programming environment that includes an extensive set of *programming support tools*. These will be needed to provide guidance in a number of forms. Their claim is that Parallelizing compilers cannot always perform well without assistance from the user. The programmer may play an important role, informing the system via *assertions* of global relationships (some of which may be due to high-level properties of the algorithm) that an automatic state-of-the-art tool cannot detect. One of the main reasons of this is the indecidability or intractability of many relevant problems and the lack of adequate heuristics for handling them.

2.3.3.8 Hierarchical Partitioning

Here the program to be partitioned is represented by a hierarchical graph. A hierarchical graph is one for which the nodes could contain subgraphs. These subgraphs could have nodes which in turn contain subgraphs. This goes on recursively and without any limit. IF1 and IF2 are examples of hierarchical graphs.

The hierarchical partitioning procedure is a recursive one, which takes as argument the program graph, and is then recursively applied on the subgraphs. Hence, the partitioning is done starting from the lowest-most level subgraphs and goes on to the next higher level subgraphs, until we reach the original program graph. The partitioning method proposed by Sarkar [48] is an example of a hierarchical partitioning procedure.

2.3.4 Scheduling Issues

Once the program code has been partitioned into parallel tasks, we need to manage these tasks (i.e. managing the parallelism) for efficient parallel execution on the multiprocessor. The management of these tasks is called *scheduling.* Stated more formally, scheduling consists of assigning the tasks that result from partitioning the program (i.e. exposing the parallelism) to the available processors so as to minimize the parallel execution time. For each task (or process), we have to decide on *when* to execute that task and *where*[11] (i.e. on which processor) to execute it. Clearly, for the tasks that are assigned to the same processor, we have to decide on the order of execution of these tasks.

The most obvious answer to the *when* question is to execute tasks as soon as their inputs are available[12] [19, 20]. However, this could cause our system to saturate and hence deadlock. This is so because some parent tasks that are being executed might not find any available PEs on which to spawn their child subtasks to give them the values needed to complete. Thus, a *throttle* is needed.

As was mentioned before, for DMMs, it is particularly important to reduce both inter-PE communication and load imbalance. Scheduling is necessary to achieve a good processor utilization and to optimize inter-PE communication in the target multiprocessor.

It is well known that finding the optimal scheduling is an NP-complete problem. Although scheduling is an old, notorious problem with numerous versions and has attracted the attention of many researchers in the past, the results known to date offer little help when dealing with real parallel machines. The complexity of the scheduling problem has led the computing community to adopt heuristic approaches for each new machine. For maximum performance, the problem should be entirely left to the user[13]. However, this makes programming the multiprocessor very user unfriendly, tedious and very time consuming. Worse yet, this work cannot be ported to other parallel machines if the need arises to run the same

[11]This is called task distribution.

[12]This is a due to the functional nature of our intermediate form.

[13]This involves hand-coding and manually inserting system calls in the program.

problem in a different machine. On the other hand, it is very difficult or impossible to find a universal solution for problems such as scheduling, minimization of interprocessor communication, and synchronization. The reason is that these problems are architecture dependent. A realistic goal would be to find systematic and automatic solutions for the scheduling problem for large classes of machine architectures. For some existing scheduling methods, refer to [46, 48, 42, 27].

When performing the scheduling, there are several factors that have to be taken into account. Some of these factors are the communication and scheduling overhead, and the task granularity. Low task granularity results in high scheduling and inter-PE communication overhead.

The scheduling methods can be broadly distinguished into three classes: *static*[14], *dynamic*[15] and *hybrid static/dynamic*.

2.3.4.1 Static Scheduling

In static scheduling, processors are assigned tasks at compile time, before execution starts. When program execution starts, each processor knows exactly which tasks to execute.

When static scheduling is used, there is no overhead due to run-time scheduling; all inter-PE synchronization and communication is directly compiled in the code. Both task scheduling overhead and load balancing overhead are eliminated. Further, there is a greater opportunity to optimize inter-PE communication when the processor assignment is known at compile-time. A global compile-time analysis reduces communication overhead for the entire program. Such an analysis cannot be done on the fly at run-time. However, the efficiency of the scheduling is questionable in this case, because many of the facts regarding the behavior of the program, such as memory access patterns are only known at run-time[16]. Also, many "adaptive" applications change their access patterns and data locations over their execution lifetime. For these kind of programs, a compile-time

[14]Static scheduling is also called compile-time scheduling.

[15]Dynamic scheduling is also called run-time scheduling.

[16]Some of the examples of this are: array subscripts which are functions with unknown values at compile-time, conditional statements, etc.

solution might lead to a very inefficient execution, and therefore a run-time or a hybrid run-time/compile-time approach should be used. For programs with fairly predictable execution times at compile-time, this approach could be very efficient.

2.3.4.2 Dynamic Scheduling

A scheduling scheme is called dynamic when the actual processor allocation is performed by hardware or software mechanisms during program execution (i.e. at run-time). Therefore, during dynamic scheduling, decisions for allocating processors are taken on-the-fly for different parts of the program, as the program executes.

With dynamic scheduling, the run-time overhead becomes a critical factor and may account for a significant portion of the total execution time of the program. For static scheduling, the compiler or an intelligent preprocessor is responsible for making the scheduling decisions. However, for dynamic scheduling, this decision must be made at run-time in a case by case fashion, and the time spent for this decision-making process is reflected in the program's total execution time. Note that we can have the scheduler make scheduling decisions for a chunk of the program, while other parts of the program are already executing on the processors. This may reduce the overhead but does not eliminate it. The large overhead of run-time analysis necessitates very simple scheduling algorithms.

For shared memory multiprocessors, dynamic scheduling is much easier to deal with. Shared memory implementations of SISAL uses a shared ready queue to enqueue all the tasks that are ready to be executed[17]. This queue is allocated from shared memory, and therefore is accessible to all processors. Whenever a processor becomes idle, it fetches a task to execute from the ready queue. A throttle has not been needed for this approach. The advantage of this scheme is that there is no need for any dynamic load balancing algorithm, since it is done implicitly by the use of the shared ready queue. The disadvantage is that the queue is a shared resource that must be accessed using a critical section. This results in *contention* for the shared resource, which causes run-time overhead that consists of the time

[17]These are either newly created tasks, or tasks that were previously blocked (e.g. waiting for memory to become available or some value to be computed) and become unblocked.

required to execute the *lock* protocol, and the time the process has to wait for the lock (in case it has to wait).

Dynamic scheduling for DMMs can be performed through a *central* control unit[18] or it can be *distributed.* Dynamic scheduling through a central control unit usually creates too much communication overhead and results in a bottle-neck at the processor where the central control unit executes. Hence it is usually more efficient to adopt a self-scheduling scheme. A special case of scheduling through distributed control units is *self-scheduling.* As implied by the term, there is no single control that makes global decisions for allocating processors, but rather the processors themselves are responsible for determining what task to execute next. For an example of a self-scheduling method called guided self-scheduling[19], refer to [42]. Another way to implement distributed dynamic scheduling is to establish a fixed spawning pattern for each node[20]. Here, the processor executing a task uses a predefined method to decide on where (i.e. on which processor) to execute the child tasks. This method needs some load balancing scheme to try to keep the work evenly distributed on all processors. In fact, when we use a dynamic scheduling mechanism, dynamic load balancing is usually needed. One more scheme to implement distributed scheduling is to adapt the shared ready queue idea of the shared memory implementation of SISAL. Here, each processor is given its own *private* ready queue. Each processor monitors its own private ready list, which now can be done without having to obtain exclusive access, and executes any task that arrives. Some mechanism is still needed to decide to the queue of which processor to send a task when it is ready to execute (for instance when a child task is newly created by the parent, or when a previously blocked task is ready to resume execution). Unlike for the shared memory case, we now don't need to obtain critical section locks to access the ready queue. Hence, the overhead for contention is eliminated, and we now allow for a scalable number of processors to be employed. However, we now no longer have implicit load balancing capabilities, and therefore some dynamic load balancing algorithm is

[18]Some people call this unit the *arbitrator*.

[19]This method applies to shared memory machines and is restricted to arbitrary nested parallel loops.

[20]This is the method that was first used for shared memory implementation of SISAL.

needed, for it is now possible for the system load to become unbalanced. For a detailed implementation of this scheme, refer to [23].

2.3.4.3 Hybrid static/dynamic scheduling

We can also have a *hybrid static/dynamic* scheduling. In this case, we start with an initial scheduling at compile-time, and allow the assignment of tasks to PEs to change at run-time (task migration), if we know that this will give us a better performance. For example, we might have to do some task migration in order to balance the load.

During run-time, it is not possible to have all the knowledge about the topology of the program. Compiler support of some form is required for that. Therefore, a hybrid dynamic/static scheduling is probably the best approach to solve this problem. In this scheme, the compiler helps the arbitrator or the processors[21] in making a scheduling decision.

The hybrid approach could benefit us from the low overhead of static scheduling and the better task assignment of dynamic scheduling.

2.3.5 Distribution of Data

Data distribution is the task of dividing the data structures that a program uses among the memory elements so as to minimize the total execution time of the program. This is equivalent to distributing the data structures so that the number of remote references is minimal. Therefore, it is necessary to keep the task distribution and the data distribution closely tied, and keep these two aligned at all times. Ideally, we want all memory accesses to be local. Clearly, this is not possible.

The data distribution problem is a very difficult and complex one. In fact, finding the optimal data partition is an NP-complete problem. Nevertheless, if good performance is to be achieved, this problem has to be addressed. Since the optimal partition is very hard to find, we have to settle for heuristic methods that give us performance as close to optimal partition as possible. An appropriate

[21]depending on whether we use a central or a distributed scheduling scheme.

	static	hybrid	dynamic
implicit			
hybrid			
explicit			

Figure 2.2: Classification of data distribution methods

heuristic method for automatically determining a nearly optimal data partition has yet to be found.

2.3.5.1 Classification of data distribution methods

Usually, the data partitioning methods are classified according to whether they are implicit or explicit, compile-time or run-time (see figure 2.2).

- *compile-time data distribution*: The distribution of data structures is done at compile-time. This is also called *static* data distribution. By static we mean that the distribution remains the same throughout the life-time of the program. There are several ways this is done, and in what follows we list a few of them:

  1. The compiler uses some **distribution function** to partition the data. Distribution functions can be either predefined or user supplied. Typically, a distribution function has many parameters that control the way data structures are partitioned. Ideally these parameters are chosen automatically by the compiler, after doing some formal analysis, such as analyzing the access patterns, or with the help of run-time profiles [Sar89]. In the case of run-time profiles, the compiler watches several characteristic runs and notes the distribution patterns used for those runs. The compiler then selects a distribution function that will come

closest to the observed reference behavior. Note that this approach is inefficient if the profile runs are not characteristic of the actual reference patterns, or if the reference patterns vary with the input data. Due to the complexity of this task, many compilers rely on some user intervention, either in the form of language extensions or pragmas (compiler directives). For an example of that, refer to [26].

2. The compiler uses a random distribution of data structures. In this case, the compiler tries to make the distribution even across all PEs without regard to the access patterns. Naturally this is a very simple scheme. But its performance could be unacceptable.

- *Hybrid compile-time/run-time data distribution:* The term hybrid refers to the possibility of changing the distribution associated with the data-structure at run-time. An initial distribution is done by the compiler. Then during run-time, a **redistribution (re-mapping)** of the data is done in order to reduce the remote accesses. This method is efficient in case we have many compile-time unknowns, such as array subscripts which cannot be determined at compile time. Also for programs where the access pattern changes during the execution of the program, it might not be possible to find a mapping of the data structures that will result in minimal remote references before and after the access pattern changes. In other words, the best mapping that gives the least amount of remote references for the program up to the point just before the access pattern changes might give a big number of remote references during the execution from the point when the access pattern changes and on. For example, for the FFT algorithm, the access pattern changes over the iteration space. For this kind of problems, re-mapping of the data to recapture local references could result in some improvements. The problem with this method is that the analysis required to determine when re-mapping is worth performing is a very difficult one. Also for DMMs for which the inter-PE communication is expensive, this method might cause some tremendous run-time overhead due to the extra inter-PE communication caused by moving the data around during the re-mapping, and updating the descriptors of the data structures that are re-mapped to record the new distribution. Hence, it is important that the communication incurred by the redistribution of the data (to minimize communication during a computation) does not exceed the communication overhead which that redistribution was intended to reduce.

- *Run-time data distribution*: Here all decisions regarding the distribution of data are done at run-time. This is also called *dynamic* data distribution. One way to do this is to make the decisions regarding distribution of data, functions of some parameters that are only known at run-time.

- *Explicit data distribution*: The programmer controls the data distribution explicitly[22]. In this case, the programmer explicitly inserts the appropriate inter-PE communication primitives. All of this work is done "by hand" and contradicts the goal of raising programming to a higher level of abstraction.

- *Hybrid Explicit/Implicit data distribution*: In this method, we use compile-time, hybrid compile-time/run-time, or run-time data distribution, with some user intervention. More precisely, the programmer gives some hints regarding how distribution of data should be done, and the compiler or the run-time system uses those hints to decide how to distribute the data structures. For example, the hints could be in the form of some compiler directives, pragmas or assertions. This method put some burden on the programmer, but has more potential for success than the purely automatic data partitioning methods. This is so because the programmer could be very aware of the data access patterns of his program, and therefore could know about the optimal or near-optimal partitioning of the data structures in the program. Some researchers feel that the compiler on its own will not be able to choose an efficient data decomposition for all programs, and therefore it needs some additional information from the user. Fortran D [26] is an example of a language that requires the programmer to specify the data layout of the program. Then the compiler uses that information

[22]Usually, the programmer uses an explicit imperative programming method. Hence there is no compiler that further processes the code to do the parallel processing tasks (code and data partitioning, scheduling, etc.), since the programmer does all of these tasks explicitly.

to distribute the data structures (arrays) to the processing elements, and insert the appropriate message passing primitives.

- *Implicit data distribution:* Using the implicit data distribution method, the compiler or the run-time system is responsible for all the partitioning of the data structures, without any help from the user. Usually, it is quite difficult to design a system that produces very efficient data partitioning, unless some user intervention is available. Note that this method could be compile-time, run-time, or a hybrid of the two.

2.3.5.2 Using the Shared Memory Programming Paradigm for DMM's

- **Description**: In this method, a shared memory programming model is supported on top of the distributed memory architecture. This is called *Distributed Shared Memory* (DSM) or *Virtual Shared Memory* (VSM) system. The compiler or programmer is provided with a shared memory abstraction, and a set of primitives for allocating and accessing shared data structures within a virtual address space. The programmer (or the compiler) assumes that there is a contiguous, single address space (this is a *virtual* space) shared by all PEs in the network and uses this virtual memory to store and retrieve the data structures in the program. All the memory access routines in the program are done with respect to the virtual memory. Then, it is the job of some software interface to map the virtual space onto the distributed physical memories. All message passing required for accessing remote values is handled *implicitly* by this interface, through the use of a *message passing abstraction.* This message passing abstraction provides an abstract interface to the host operating system. Each data structure allocated to the virtual space receives a contiguous set of virtual addresses shared among all the PEs[23]. Note that the interface will be nothing more than the library of memory access routines called by the user program. This library of routines is responsible for the mapping between the virtual memory and the physical

[23]In fact, data can be represented in two ways: as a local memory variable (i.e. stored in the local memory of the PE in question) or as a shared addressing space variable (i.e. stored in the virtual shared memory).

memories, and all the memory management needed.
This library will be part of the run-time system, and hence the method described here is called *run-time support for a single addressing space.*

- **Advantages:** The main advantage of this scheme is that the library created can be written as a stand-alone module in a generic (language independent) way. This module can therefore be plugged into any other run-time system for compilers of other languages. With some minor modifications to the message passing routines used, this library could even be ported to other DMMs.
  Another advantage is that this *message passing abstraction* makes the task of the programmer (or compiler) much simpler, since all message passing routines and data distribution will be handled by the library of memory access routines. However, the way the data distribution is done should be dictated by the compiler (or the programmer), since these library routines don't have any information regarding the behavior of the program. One way to implement this is to include the mapping function as a parameter in the routine that allocates the virtual space to data structures, so that this routine can use that information to distribute the data across the PEs.

- **Implementation issue:** Using the virtual shared memory scheme has another advantage as far as the implementation goes. Instead of writing a new compiler, we can simply use the existing OSC compiler with some miner modifications[24].

- **Disadvantages**: The main disadvantage of this method is the run-time overhead caused by the translation of virtual addresses to physical ones.

- **Address Translation**: When the compiler encounters an array access A[i], it translates it to some code in the output program, that computes the address of the element A[i] and then fetches that location. In our case, this address is a virtual one. Therefore, we have to translate this virtual address to the corresponding physical one, which for DMMs constitutes of a

[24]OSC was written for shared memory multiprocessors.

processor number and the local physical address on that processor. Clearly this translation process is done by the library routines that manage the virtual address space. One way to do the translation is to have a table that has one entry for each virtual address and the corresponding physical address. We need to replicate this table in all PEs in order not to create too much communication traffic. Although this method makes the translation process very cheap in terms of time, since it would require one access of this table, it is extremely costly in terms of space and therefore not feasible. The other way to do the translation is to use the information regarding the virtual space allocated to the data structure and the way it is distributed. From this, we can use some formulas to deduce the virtual address. This method is not costly in terms of space, since we don't have to store the physical addresses corresponding to virtual ones, however it is very costly in terms of time because of the computations involved. There are ways to optimize the address translation using the second method that we mentioned. For instance, for virtual addresses corresponding to local physical ones (physical addresses on the current PE), we can use some tricks to recognize these virtual addresses and get a direct mapping to physical ones, without using any computations [21, 23].

- It is clear that using this method does not affect the inter-PE communication overhead. Very little extra memory space is needed to record the required information for the translation of virtual addresses to physical ones for each array.

- This distributed shared memory method was used to implement SISAL on DMMs[25] [21, 23]. The results were not very impressive however. Some of the problems were the run-time overhead introduced by the VISA calls[26], and the fact that multi-dimensional arrays were not implemented as true multi-dimensional arrays[27].

[25]VISA is the name of the library that was designed.

[26]The main overhead was caused by the address translation.

[27]SISAL and its intermediate forms IF1 and IF2 assume that all arrays are one-dimensional. Multi-dimensional arrays are implemented as arrays of arrays.

- In order to make this method more attractive, we have to devise some address translation scheme that does not involve many computations. We should be able to afford to sacrifice some memory space for less time, provided that the space traded for time is not too large.

2.4 Existing Code Partitioning Methods

Existing work regarding the partitioning problem either considers a specific application and try to come up with an efficient partitioning scheme for it (i.e. no automatic partitioning), or come up with a general solution (automatic partitioning) that is too simple and therefore not efficient (e.g. exploits only one kind of parallelism level). The partitioning methods that belong to the second class and that have been implemented in real systems exploit only certain levels of parallelism, with certain granularity. Some methods use syntax-based partitioning, some others use function-level partitioning, some systems exploit instruction level parallelism leading to very fine grain size, etc. We believe that the granularity of parallelism should depend on the particular application that we are solving and on the target machine.

The best partitioning scheme that we are aware of is the one proposed by Sarkar [48, 47, 46, 49] and is described briefly next. As was mentioned before, this method is too simple.

Sarkar's Partitioning Method

Sarkar considers both static and dynamic scheduling, and only static program partitioning. His partitioning method for static scheduling differs from the one for dynamic scheduling. Since we assume static scheduling, we only describe Sarkar's partitioning method for static scheduling.

Algorithm

In this section, we describe the overall algorithm without going into any details.

1. All communication edges in the program graph are sorted in decreasing order of their communication cost.

2. For each edge e in the graph, starting with the one that has the highest communication cost, we merge e if the merger does not cause an increase in the parallel execution time (the critical path length of the graph).

Time Complexity

Let E be the number of edges and N be the number of nodes in the program graph.
Sorting the E edges takes $O(E^2)$ time in the worst case.
The algorithm requires that the parallel execution time be computed for each iteration (i.e. each edge examined) of the algorithm. The parallel execution time can be computed by traversing the graph. In the worst case, this takes $O(E+N)$ time. Since there are E edges in the graph (i.e. E iterations in the algorithm), this will take $O(E(E+N))$ time.
Hence the overall time complexity of the algorithm is $O(E(E+N))$. We will show later on in this thesis that $E < N^2$. Therefore, the time complexity can be written as $O(E.N^2)$.

Chapter 3

The Partitioning Problem

3.1 Assumptions

- We assume that we have a weighted Directed Acyclic Graph (DAG) representation of the program. The DAG is flat (no hierarchical graphs such as IF1 and IF2, the intermediate graph representation of SISAL), the nodes represent instructions (primitive or compound statements), and the edges represent data. The assumption that we don't have any hierarchical graphs makes the analysis simpler, and it allows us not to use hierarchical partitioning algorithms which are in general more complex.

- One way to get the input DAG described earlier is to use IF1 or IF2, and a variation of Sarkar's graph expansion method[1] ([47, 46, 48, 49]) to do some preprocessing to get a non-hierarchical graph. Doing this, some compound nodes (compound nodes are nodes that contain subgraphs, and correspond to compound statements in SISAL) in the original graph (IF1 or IF2) will be nodes in the final flat DAG used as input to our analysis. In this case, all subgraphs of the compound nodes will be transparent, and we only look at the functionality of the node (i.e. given some input, we are interested in what output the nodes generates).

  The advantage in using IF1 or IF2 is that we can use the SISAL compiler

[1]Sarkar's graph expansion method is better suited for shared memory multiprocessors. It does not take into account the high cost of inter-PE communication for DMMs, and therefore is in general not efficient when used to target DMMs.

to generate the intermediate form (IF1 or IF2), then preprocess this graph to get the final input DAG used by our partitioning analysis.

- We assume that the DAG is applicative and therefore we don't have any side effects when we execute it. The data carried by the edges represent mathematical variables and not memory cells. This makes the detection of parallelism straight-forward.
- All the inputs of the DAG are assumed to be ready when program starts execution.
- The target DMM is assumed to have point to point communication links (no busses).
- The output of our partitioning analysis is the input to the scheduling phase. The best scheduling method known so far is the DSC (Dominant Sequence Clustering) designed by Tao Yang ([18, 58, 59]).

3.2 Definitions

Code Partitioning: Given a multiprocessor M and a DAG g to be executed on M, a partition of g is a set $\Pi = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where each τ_i is a non-empty set consisting of nodes in g that have to be executed on the same PE,[2] and where all the τ_i's are disjoint, and their union forms all the nodes that belong to g. Each set τ_i is called a *task*.

Definition: The *trivial* partition is the one for which each task τ_i is a singleton set (i.e. this is the partition that puts each node in a single task). All other partitions are said to be *non-trivial*.

Input and Output Nodes: Given a DAG g, an *input (entry)* node in g is any node for which an input edge carries an input data, and an *output (exit)* node in g is any node for which an output edge carries an output data.

[2]Nodes belonging to the same τ_i could be independent of each other, and therefore could in theory be executed in parallel.

There is no predecessor node to an input node and there is no successor node to an output node.
Input nodes are also called *root* nodes, and output nodes are also called *leaf nodes.*

Execution Path: Given a DAG g, an execution path of g is any path from an input node to an output node.

Critical Path: Given a DAG g, a critical path of g is any longest execution path in g.
The critical path length of g is the length of a critical path of g.
Let P_{crit} := Critical path of the DAG.
Let CPL := Critical Path Length of the DAG.

Independent Nodes: Given a DAG, two nodes are dependent if and only if there is a path between them. Otherwise they are independent.
We define $\|$ as the independency relationship, and $\perp$ as the dependency relationship.
$n_1 \| n_2$ means n_1 and n_2 are independent.
$n_1 \perp n_2$ means n_1 and n_2 are dependent.

Independent Sets

1. Given a DAG g, an independent set is a set of nodes in g in which each pair of nodes are independent (i.e. all nodes are pairwise independent).
2. Given a DAG g and a set S of nodes belonging to g, an independent set of S is an independent set which is contained in S.

Dependent Sets

1. Given a DAG g, a dependent set is a set of nodes in g in which each pair of nodes are dependent (i.e. all nodes are pairwise dependent).
2. Given a DAG g and a set S of nodes in g, a dependent set of S is a dependent set which is contained in S.

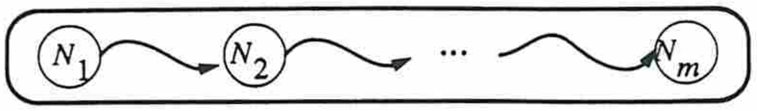


Figure 3.1: Path from N_i to N_{i+1} $(1 \leq i \leq m-1)$

Remarks

- If a set is not independent, it does not necessarily imply that it is dependent.
- Any path in the DAG constitutes a dependent set.

Parallel Sets

1. Given a DAG g, a parallel set is an independent set which is not contained in any other independent set.
2. Given a DAG g and a set S of nodes belonging to g, a parallel set of S is an independent set of S which is not contained in any other independent set of S.

Anti-Parallel Sets

1. Given a DAG g, an anti-parallel set is a dependent set which is not contained in any other dependent set.
2. Given a DAG g and a set S of nodes belonging to g, an anti-parallel set of S is a dependent set of S which is not contained in any other dependent set of S.

Theorem: Let g be a DAG.
Let $n_1, n_2, \ldots, n_m$ be nodes in the graph.
$\{n_1, n_2, \ldots, n_m\}$ $(m \geq 2)$ is a dependent set $\Longleftrightarrow$
$\exists$ a path containing nodes $n_1, n_2, \ldots, n_m$ (the path could contain other nodes as well).

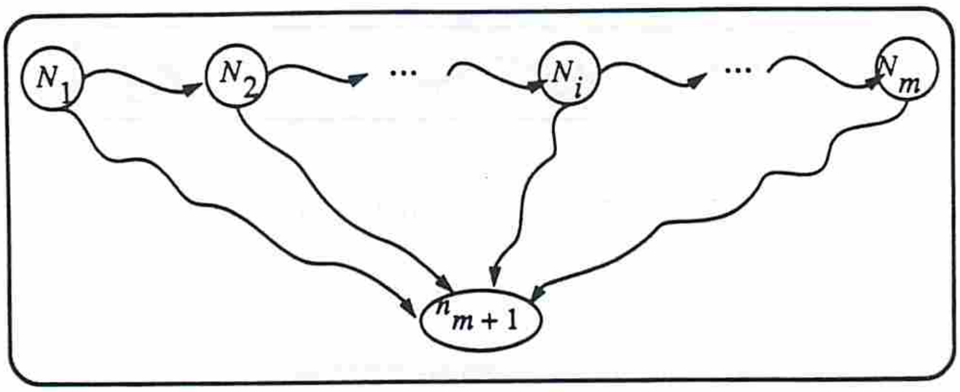


Figure 3.2: Path from N_i to n_{m+1} $(1 \leq i \leq m)$

Proof

1. $\exists$ a path containing nodes $n_1, n_2, \ldots, n_m$

   $\Rightarrow$

   $\{n_1, n_2, \ldots, n_m\}$ is a dependent set:

   Trivial.

2. $\{n_1, n_2, \ldots, n_m\}$ is a dependent set

   $\Rightarrow$

   $\exists$ a path containing nodes $n_1, n_2, \ldots, n_m$:

   We use proof by induction.

   Base Case: $m = 2$

   $n_1 \perp n_2 \Rightarrow$ there is a path from n_1 to n_2 or from n_2 to n_1.

   Induction Step: We assume that the property is true for $m \geq 2$.

   Let's prove that the property is true for $m + 1$.

   Assume that $\{n_1, n_2, \ldots, n_m, n_{m+1}\}$ is a dependent set

   $\Longrightarrow$

   $\{n_1, n_2, \ldots, n_m\}$ is a dependent set

   $\Longrightarrow$

   $\exists$ a path containing nodes $n_1, n_2, \ldots, n_m$

   $\Longrightarrow$

   There is a path from N_i to N_{i+1} $(1 \leq i \leq m-1)$, where $N_i \in \{n_1, n_2, \ldots, n_m\}$

$(1 \leq i \leq m)$, and all the N_i's are different from each other (see figure 3.1).

$n_{m+1} \perp N_1 \Rightarrow$ there is a path from N_1 to n_{m+1} or from n_{m+1} to N_1.

If there is a path from n_{m+1} to N_1, then there is a path containing nodes $N_1, N_2, \ldots, N_m, n_{m+1}$

$\Longrightarrow$

$\exists$ a path containing nodes $n_1, n_2, \ldots, n_m, n_{m+1}$.

Now assume that there is a path from N_1 to n_{m+1}.

$n_{m+1} \perp N_2 \Rightarrow$ there is a path from N_2 to n_{m+1} or from n_{m+1} to N_2.

If there is a path from n_{m+1} to N_2, then there is a path containing nodes $N_1, N_2, \ldots, N_m, n_{m+1}$

$\Longrightarrow$

$\exists$ a path containing nodes $n_1, n_2, \ldots, n_m, n_{m+1}$.

Now assume that there is a path from N_2 to n_{m+1}.

$n_{m+1} \perp N_3 \Rightarrow$ there is a path from N_3 to n_{m+1} or from n_{m+1} to N_3.

If there is a path from n_{m+1} to N_3, then there is a path containing nodes $N_1, N_2, \ldots, N_m, n_{m+1}$

$\Longrightarrow$

$\exists$ a path containing nodes $n_1, n_2, \ldots, n_m, n_{m+1}$.

Now assume that there is a path from N_3 to n_{m+1}.

We keep doing this reasoning until we reach node N_m.

$n_{m+1} \perp N_m \Rightarrow$ there is a path from N_m to n_{m+1} or from n_{m+1} to N_m.

If there is a path from n_{m+1} to N_m, then there is a path containing nodes $N_1, N_2, \ldots, N_m, n_{m+1}$

$\Longrightarrow$

$\exists$ a path containing nodes $n_1, n_2, \ldots, n_m, n_{m+1}$.

Now assume that there is a path from N_m to n_{m+1} (see figure 3.2)

$\Longrightarrow$

There exists a path containing nodes $N_1, N_2, \ldots, N_m, n_{m+1}$

$\Longrightarrow$

$\exists$ a path containing nodes $n_1, n_2, \ldots, n_m, n_{m+1}$.

Hence the property is true for $m + 1$.

Theorem: Let g be a DAG.

Let $n_1, n_2, \ldots, n_m$ be nodes in the graph.
$\{n_1, n_2, \ldots, n_m\}$ $(m \geq 2)$ is an anti-parallel set
$\Longrightarrow$
$\exists$ a path containing *only* nodes $n_1, n_2, \ldots, n_m$.

Proof

Assume that $\{n_1, n_2, \ldots, n_m\}$ $(m \geq 2)$ is an anti-parallel set. Then $\{n_1, n_2, \ldots, n_m\}$ is a dependent set. Therefore, from a theorem stated earlier, there exists a path p containing nodes $n_1, n_2, \ldots, n_m$. Hence, there is a path $p_{i,i+1}$ from N_i to N_{i+1} $(1 \leq i \leq m-1)$, where $N_i \in \{n_1, n_2, \ldots, n_m\}$ $(1 \leq i \leq m)$, and all the N_i's are different from each other (see figure 3.1).
We claim that path $p_{i,i+1}$ is equal to edge (N_i, n_{i+1}), $1 \leq i \leq m-1$.
To see why this is the case, assume that the claim is incorrect.
Then, $\exists$ a node N_j, $j \neq i$ and $j \neq i+1$ such that $n_j \in p_{i,i+1}$. Clearly $N_j \neq N_k, 1 \leq k \leq m$, otherwise we will have a cycle in path p. Since $N_j \perp N_k, 1 \leq k \leq m$, then $\{N_1, N_2, \ldots, N_m, N_j\}$ is a dependent set. Hence, $\{n_1, n_2, \ldots, n_m, N_j\}$ is a dependent set. This means that $\{n_1, n_2, \ldots, n_m\}$ is not an anti-parallel set, which contradicts our original assumption. Therefore, our claim is correct, which means that path p contains only nodes $n_1, n_2, \ldots, n_m$.

Remark: Given a DAG g.
$\exists$ a path containing *only* nodes $n_1, n_2, \ldots, n_m$ $(m \geq 2)$
$\not\Rightarrow$
$\{n_1, n_2, \ldots, n_m\}$ is an anti-parallel set.
To see why this is the case, assume that there exists a path p containing only nodes $n_1, n_2, \ldots, n_m$. Then there could exist a path p' which contains p, $p' \neq p$ (see figure 3.3 for 2 examples of such a situation), in which case there exists at least a node n, $n \neq n_i, 1 \leq i \leq m$, such that $n \perp n_i, 1 \leq i \leq m$. Hence, $\{n_1, n_2, \ldots, n_m, n\}$ is a dependent set. Therefore, $\{n_1, n_2, \ldots, n_m\}$ cannot be an anti-parallel set.

Theorem: Given a DAG g.
Let $n_1, n_2, \ldots, n_m$ be nodes in the graph.

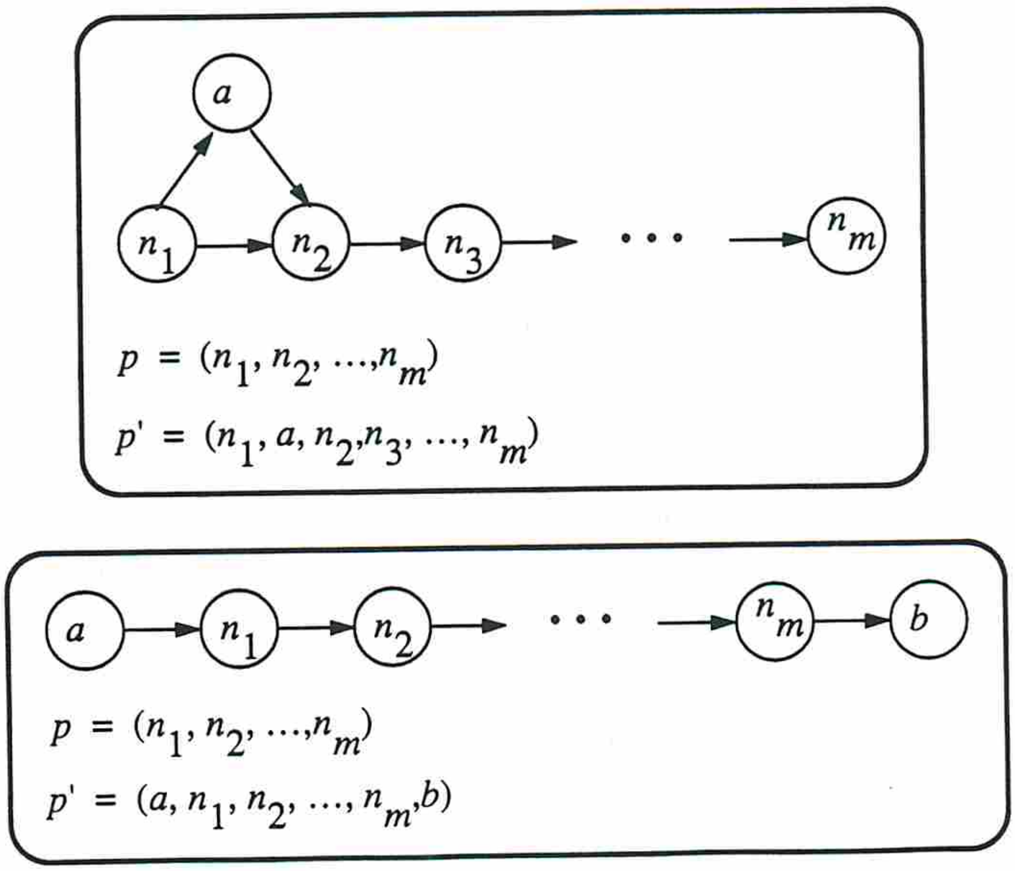


Figure 3.3: 2 examples where $\{n_1, n_2, \ldots, n_m\}$ cannot be an anti-parallel set

$\exists$ a *unique* path p containing nodes $n_1, n_2, \ldots, n_m$, $(m \geq 2)$
AND
p contains *only* nodes $n_1, n_2, \ldots, n_m$

$\Longrightarrow$

$\{n_1, n_2, \ldots, n_m\}$ is an anti-parallel set.

Proof

Assume that:
There exists a *unique* path p containing nodes $n_1, n_2, \ldots, n_m$, $(m \geq 2)$
AND
p contains *only* nodes $n_1, n_2, \ldots, n_m$

Clearly, $S = \{n_1, n_2, \ldots, n_m\}$ is a dependent set.
Let's assume that S is not an anti-parallel set.
Then there exists at least one node n, $n \neq n_i, 1 \leq i \leq m$, such that $\{n_1, n_2, \ldots, n_m, n\}$ is a dependent set. From a previous theorem, this implies that there exists a path p' containing nodes $n_1, n_2, \ldots, n_m, n$. Since p is the unique path containing nodes $n_1, n_2, \ldots, n_m$, then $p = p'$. But p contains only nodes $n_1, n_2, \ldots, n_m$, which contradicts the fact that $n \in p$. Hence, our assumption that S is not an anti-parallel set is wrong.

3.3 Some Properties

$\perp$ Relationship

1. The relationship is not reflexive.

2. The relationship is symmetric.

3. The relationship is not transitive.
   In figure 3.4-a, $n_1 \perp n_2$, $n_2 \perp n_3$, but $n_1 \parallel n_3$.

$\parallel$ Relationship

1. The relationship is not reflexive.

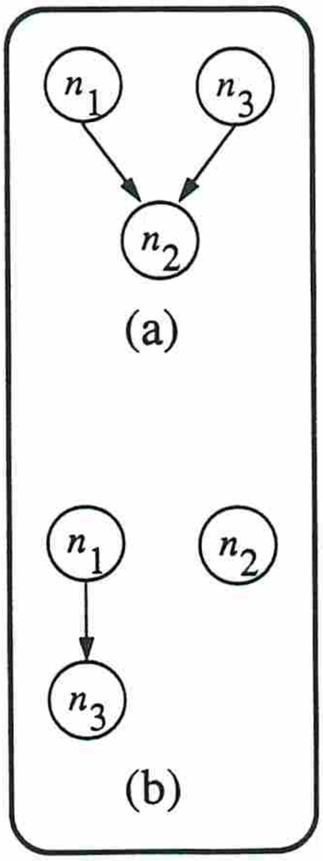


Figure 3.4: $\|$ and $\perp$ are not transitive

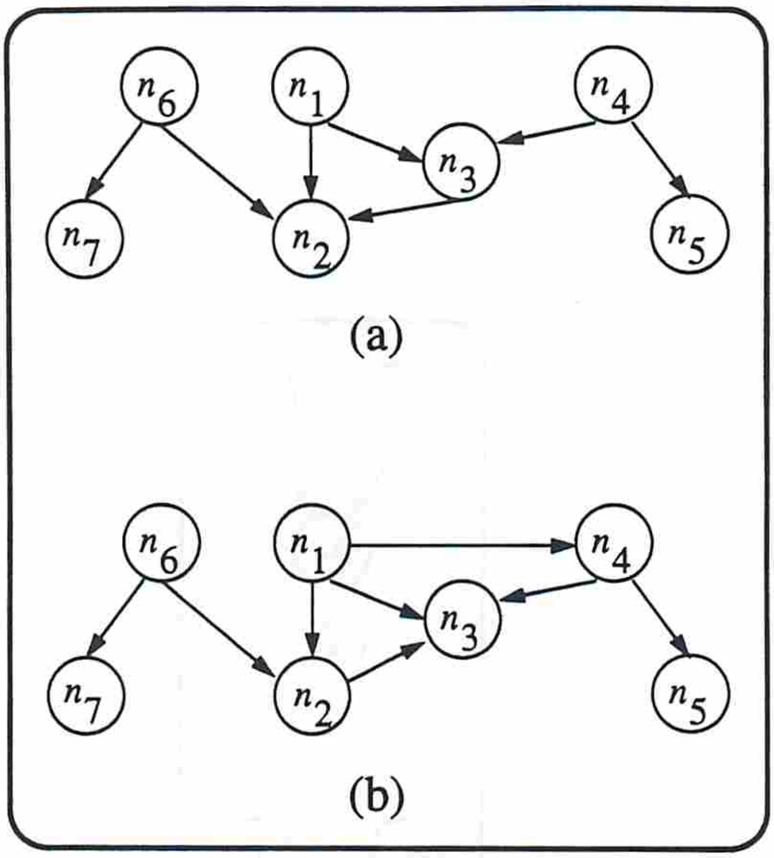


Figure 3.5: Number of elements in parallel sets

2. The relationship is symmetric.
3. The relationship is not transitive.
   In figure 3.4-b, $n_1 \parallel n_2$, $n_2 \parallel n_3$, but $n_1 \perp n_3$.

Parallel Sets

Consider a DAG g.
The parallel sets don't always have the same number of elements.
In figure 3.5-a, the parallel sets are: $\{n_6, n_1, n_4\}$, $\{n_6, n_1, n_5\}$, $\{n_6, n_3, n_5\}$, $\{n_7, n_2, n_5\}$, $\{n_7, n_1, n_4\}$, $\{n_7, n_1, n_5\}$, $\{n_7, n_3, n_5\}$. In this case, all parallel sets have the same number of elements.

In figure 3.5-b, the parallel sets are: $\{n_6, n_1\}$, $\{n_6, n_4\}$, $\{n_6, n_5\}$, $\{n_7, n_2, n_4\}$, $\{n_7, n_2, n_5\}$, $\{n_7, n_1\}$, $\{n_7, n_3, n_5\}$. In this case, not all parallel sets have the same number of elements.

Anti-Parallel Sets

Consider a DAG g and a set S of nodes belonging to g.
In general, the set S has zero or more anti-parallel sets, and a set S_{ind} of zero or more nodes which don't belong to any dependent set[3].
Assume that the anti-parallel sets are $S_1, S_2, \ldots, S_k$, and that S_{ind} is $\{n_1, n_2, \ldots, n_m\}$, where[4]
$k \geq 0$ and $m \geq 0$.
In general, the anti-parallel sets are not necessarily disjoint. Furthermore, 2 nodes belonging to 2 different anti-parallel sets could be dependent.
For an example of this, consider figure 3.5-b.
Let $S = \{n_1, n_2, \ldots, n_7\}$.
The anti-parallel sets are: $\{n_6, n_7\}$, $\{n_6, n_2, n_3\}$, $\{n_1, n_2, n_3\}$, $\{n_1, n_4, n_3\}$, $\{n_1, n_4, n_5\}$.
In this case $S_{ind} = \emptyset$.
Figure 3.5-a shows another example.
Let $S = \{n_1, n_2, \ldots, n_7\}$.
In this case, the anti-parallel sets are: $\{n_6, n_7\}$, $\{n_6, n_2\}$, $\{n_1, n_2, n_3\}$, $\{n_2, n_3, n_4\}$, $\{n_4, n_5\}$.
Also for this case, $S_{ind} = \emptyset$.
Note that in this case, S can be written as
$S = S_1 \cup S_2 \cup S_3$, where
$S_1 = \{n_1, n_2, n_3\}$, $S_2 = \{n_4, n_5\}$ and $S_3 = \{n_6, n_7\}$. S_1, S_2 and S_3 are disjoint anti-parallel sets.
Note also that in some cases, the set S can be expressed as

[3]Each node in S_{ind} and any other node in S are independent, and clearly S_{ind} is an independent set of S. However, S_{ind} is not a parallel set.

[4]$k = 0$ represents the case where S has no dependent sets (i.e. S is an independent set), and $m = 0$ represents the case where each element in S belongs to some dependent set.

$S = S_1 \cup S_2 \cup \ldots \cup S_k \cup S_{ind}$, where the S_i's are disjoint anti-parallel sets, and S_{ind} is a set of one or more nodes which don't belong to any dependent set.

3.4 Task Graph

Definition: We define the *task dependency graph* (or *task graph* for short) of a partition to be the *directed* graph whose nodes are the tasks in the partition, and where the arcs between nodes represent the data dependency between tasks (i.e. there is an edge from task τ_i to task τ_j *if and only if* data has to be transmitted from τ_i to τ_j).

3.4.1 Defining Task Graph Weights

3.4.1.1 Node Weights

The weight of a node in the task graph is the sum of the execution times of all actors that belong to the task represented by the node.

3.4.1.2 Edge Weights

Consider an edge $e = (n_i, n_j)$ in the task graph, where node n_i represents task τ_i and node n_j represents task τ_j.
The weight of e is the total amount of data transmitted from τ_i to τ_j during one execution of the program.

3.4.2 Communication Between Tasks

Consider 2 tasks τ_1 and τ_2 in the task graph such that there is an edge from τ_1 to τ_2.
Assume that actors $a_1, a_2, \ldots, a_n$ belong to τ_1, and actors $b_1, b_2, \ldots, b_n$ belong to τ_2, and that there is an edge from a_i to b_i ($1 \leq i \leq n$) in the original DAG (the input program graph).

The question is: do we send the messages from a_i to b_i individually, or do we combine them into ONE larger message that is sent from τ_1 to τ_2?

In other words, do we wait for all messages destined from τ_1 to τ_2 to be ready and send them all together in one larger message, or do we send each message individually as soon as it is ready?

One advantage of sending messages individually is that the destination actors wait less time for their input data to arrive (as soon as an actor inside a task finishes execution, we send the outputs of the actor to their destination actors). However, as will be seen later, our execution model does not allow any partial execution of tasks. Hence, this will not be of any benefit to us. Note that if partial task execution were allowed, then sending the outputs of actors inside a task individually as soon as they are ready could be of great benefit.

One popular optimization technique used for DMMs is to combine smaller messages going to the same destination into larger ones before sending them whenever possible. This usually reduces the communication overhead. This is true because the dominant component in the cost to communicate a message between PEs is the message start-up time. The other component, mainly the delay component (the duration from the time the message is sent to the time it is received) is small compared to the message start-up component. Therefore, when we combine smaller messages into a larger one, we only have one message start-up for the new larger message, rather than several for each smaller message.

Because of the reasons mentioned above, we chose to combine all messages destined from τ_i to τ_j into one larger message.

3.5 Graph Execution Cost and Effect of Data Distribution

3.5.1 Nodes in the Input Program Graph

Two kinds of nodes:

1. RNODEs: Nodes whose execution always involves one or more memory accesses.

These are also the nodes whose execution may involve one or more remote accesses (in case the memory accessed is remote).

2. LNODEs: Nodes whose execution does not involve any memory accesses. These are also the nodes whose execution never involves any remote accesses.

Examples of RNODEs: Array manipulation nodes (ABuild, AFill, AElement, AReplace, ACatenate, AScatter, AGather).
Examples of LNODEs: Arithmetic and Boolean nodes.

3.5.2 Graph Execution Cost

1. Node computation cost.

2. Inter-node communication cost.

Node Computation Cost

1. LNODES: Given the target machine, we can determine the execution cost of these nodes.

2. RNODES: The execution cost of an RNODE depends on the PE to which the node is assigned, and the memory (i.e. memory of which PE) that has to be accessed. Therefore, the way data is distributed across the PEs affects the execution cost of RNODES.

Inter-Node Communication Cost

Given the assignment of nodes to PEs and the size of data transmitted along an edge in the graph, we can determine the cost of transmitting the data along the edge.

1. If an edge links two nodes assigned to different PEs, then the cost of transmitting data along this edge is the cost of communicating the data between the PEs[5].

[5]An edge connecting two nodes mapped to different PEs doesn't necessarily cause inter-PE communication. For more details, refer to the section regarding local and non-local edges.

2. If an edge links two nodes assigned to the same PE, then we assume that the cost of transmitting data along the edge is zero.

3.5.3 Cost Measures Needed for Partitioning Analysis

As will be seen later, during the partitioning analysis, we need to compute the CPL of the task graph. Evaluation of CPL of task graph requires knowledge of node computation cost and inter-node communication cost.

1. *Inter-node communication cost:*
   Since we assume that each node in the task graph is mapped to a different virtual PE, and that we have minimum non-zero communication overhead, then given the size of the data transmitted along an edge, we can figure out the inter-PE communication cost caused by this transmission.

2. *Node computation cost:*

   (a) LNODES: Given the target machine, we can estimate the cost.

   (b) RNODES: During the partitioning phase, we don't know the assignment of nodes to the physical PEs yet. Hence we have no way of telling whether an RNODE involves a remote access or not. Therefore, it is not possible to determine accurately the execution time of RNODEs. This is true even if we know the way the data is partitioned across the PEs.

3.5.4 Estimation of Execution Cost of RNODES

Data Distribution Procedure: We assume that it is a function of the number of PEs and the distance between each pair of PEs.
It does not depend on code partitioning and scheduling (i.e. it can be done before code partitioning and scheduling phases).

Method 1: Assume that all RNODES do local memory accesses only.
In this case, inter-PE communication can be caused by edges only.

Method 2: Apply data distribution procedure to map data to the virtual PEs to which the nodes in the graph are assigned.

Problems with Method 2

- There is a very large number of nodes in the graph. This number could be in the order of 10,000 or more, and therefore the number of virtual PEs used could be in the order of 10,000 or more. Hence, we might not have enough data to distribute across all virtual PEs.

- At each step of the partitioning algorithm, nodes in the graph are merged. Therefore, fewer virtual PEs are used, and we will have to apply the data distribution procedure again to take into account the change in the number of virtual PEs.

  One way to get around this problem is to assume that each time 2 nodes are merged, the corresponding virtual PEs PE_i and PE_j to which these nodes are assigned are replaced by another virtual PE PE_k, and all the data which was mapped to PE_i and PE_j is assigned to PE_k. For this to be possible, we need some way to keep track of this data reassignment. This could be costly in terms of time or memory space.

- Usually for DMMs, the cost to determine the physical addresses of the data used is quite high. Hence, this method could add too much to the time complexity of the compiler analysis.

3.5.5 Task Execution Model and Output Data Storage

- *Convexity Constraint:* A task receives all inputs before starting execution, then it executes to completion without interruption (i.e. no partial task execution).

  For this not to cause any deadlock situations, we have to make sure that the task graph is acyclic at all times.

- At the end of execution, all outputs are sent immediately to destination tasks.

- Output data is not stored in memory.
- Data is sent to the PE where destination task executes right away, using the *send* primitive.
- Destination task gets data by executing a *receive* command right before starting execution.

Convexity Constraint Versus Partial Task Execution

As was mentioned previously, our execution model does not allow any partial task execution. The question that arises here is: will this affect the utilization of the multiprocessor? For a typical application, there will be so many tasks ready to execute most of the time during run-time. Therefore, it is most probable that we can keep the multiprocessor busy most of the time even without allowing any partial task execution.

Using the convexity constraint, our execution model is simpler, and there is no need for context switching during run-time.

3.5.6 Existing work

- Partitioning and scheduling methods for DMMs don't take into account effect of data distribution.
- Execution of nodes is assumed not to cause any remote accesses, and therefore does not cause any inter-PE communication.
  Inter-PE communication can be caused by edges only.

3.5.7 How to model effect of data distribution in the graph?

Node Cost:

- LNODE: x , where $x :=$ computation cost.
- RNODE: (x, y) , where
  - $x :=$ computation cost,

- y := communication cost due to remote access, if there is any,
  $y := 0$, otherwise.

Problem:

During partitioning phase, y is unknown, since assignment of nodes to PEs hasn't been done yet.

3.5.8 Conclusion

- Communication can be caused by *non-local* edges and RNODEs.
- Because assignment of nodes to PEs is done after code partitioning phase, during code partitioning analysis, it won't be possible to take into account communication caused by RNODES.
- However during scheduling phase, we could use the effect of communication caused by RNODES.
  As soon as an RNODE is assigned to a PE, its corresponding y value can be determined, assuming that the data partitioning has already been done.

3.6 Parallel Execution Time (PARTIME)

Since all our analysis is done at compile-time, we have to devise some way of estimating the parallel execution time of the program at compile-time. Obviously, the only way to determine the exact execution time of the program is to run it on the multiprocessor.

3.6.1 Execution Profile Information

In recent years, execution profile information became widely used in automatic compiler optimizations. In our case, it provides us with:

- Average data sizes for all communication edges.
- Average frequency values for function calls for each function in the program.

- Average frequency values for subgraphs of parallel and compound nodes.

In order to be as accurate as possible, the only information generated from profiles is counts and sizes. Execution time costs are not used since unlike counts and sizes, they cannot be measured exactly from profiles. Execution times and communication costs are estimated from the information obtained from the profile. This information can be generated by any execution (sequential or parallel) of the program on the target machine.

A drawback of execution profiles is their sensitivity to changes in program inputs. Clearly, this could affect the optimizations done by the compiler. For this reason, it is more efficient to average the information over several inputs.

Note that if we change the target machine, then we have to regenerate the profile information even if we use the same program. This is true since the data sizes for the new machine might be different from the previous one[6].

3.6.2 Cost Assignment

An important information for our compiler analysis is the average execution time of the nodes in the graph. Here we are concerned with the *sequential* execution time of the nodes. Our approach is the same as the one in [48]. Mainly, we assume that the average execution time $f_s(n)$ of all simple nodes n is one of the target multiprocessor parameters. There are many possible techniques for estimating the execution time for simple nodes, and these techniques vary for different architectures. One simple scheme is to add the execution times of the target instructions which implement the simple node.

All average execution times of non-simple nodes are derived from f_s and from the profile information. This derivation is based on the following 3 simple rules:

1. The average execution time of a graph is the sum of the average execution times of all its nodes.

2. The average execution time of a parallel node is the product of its average number of iterations and its subgraph's average execution time.

[6]The graph frequencies remain the same however.

3. The average execution time of a non-parallel compound node is the sum of the products of each subgraph's average frequency and execution time.

3.6.3 Multiprocessor Parameters and Communication Model

The multiprocessor parameters needed for our analysis are the communication overhead between PEs and the simple node average execution time cost function f_s. $f_s(n)$ for simple node n is the execution time of n.

The communication overhead between 2 PEs is assumed to have two components [48]:

1. *Processor component*: the duration for which a processor participates in a communication. If the processor is sending data, then this is the time it takes it to write and prepare the message, before it is sent on the network. We represent this time by a function W_c. If the precessor is receiving a message, then this is the time it takes it to read the message, after it arrives to the PE from the network. We represent this time by a function R_c.
   Note that when a message is sent between 2 processors, the sum of W_c and R_c caused by this message constitutes the message start-up component (allocating buffers, copying data to or from buffers, etc.)[7].

2. *Delay component*: this is the duration from the time the message is sent on the network by the source processor (after it is written), to the time it is received by the destination processor (before it is read). This is also the fraction of communication time during which the sender and receiver precessors are free to execute other instructions. We also call this time *network component*. We represent this time by a function D_c.

D_c is a function of a 4-tuple (i, j, s, l), where i is the source processor number, j is the destination processor number $(i \neq j)$, s is the size of the message, and l is the total communication load in the multiprocessor at the time of the message

[7]The message start-up component also includes the time to execute the routing algorithm, the time to establish an interface between the local processor and the router, etc.

transfer. R_c is a function of the couple (j, s) and W_c is a function of the couple (i, s).

DMMs use message passing as a means for communication between PEs (synchronization and remote data access). The message passing protocol uses the *send* and *receive* commands. We assume that *send* is *non-blocking* and *receive* is *blocking.*

Assume that processor PE_i communicates a message to processor PE_j. Let t_1 be the time when PE_i executes the send command and t_2 be the time when PE_j executes the receive command[8]. The message arrives at PE_j at time $t_3 = t_1 + W_c + D_c$. Let's compute the time taken by PE_i and PE_j due to this communication.

Sending Processor

For processor PE_i, there will be no idle time since the *send* command is non-blocking. Thus the time taken by the sending processor due to this communication is simply W_c.

Receiving Processor

There are two possibilities:

1. If the receive command is executed at any time after t_3 ($t_2 \geq t_3$), then PE_j will never idle to wait for the message to arrive:
   $idletime = 0$.
   PE_j will spend another R_c time to read the data, and therefore the data will be available at time $t_2 + R_c$. We say that all the communication delay has been overlapped with computation in PE_j.
   The time taken by PE_j due to this communication is:
   $idletime + R_c = R_c$.

2. If on the other hand the receive command is executed before t_3 ($t_2 < t_3$), then PE_j will idle for $t_3 - t_2$ time to wait for the message to arrive:

[8]Here we assume that we have a global clock used by all PEs.

$idletime = t_3 - t_2 = t_1 + W_c + D_c - t_2 = t_1 - t_2 + W_c + D_c = \Delta t + W_c + D_c$, where $\Delta t = t_1 - t_2$ is the difference between the time when PE_i executes the send command and the time when PE_j executes the receive command. In this case, the data will be available in PE_j at time $t_3 + R_c$.
The time taken by PE_j due to this communication is:
$idletime + R_c = \Delta t + W_c + D_c + R_c$.

In conclusion, for the receiving processor, the time taken due to the communication is $idletime + R_c$.
idletime is equal to zero or $\Delta t + W_c + D_c$, depending on when the receive command was executed relative to the send command. Another expression for *idletime* is $idletime = \alpha(\Delta t + W_c + D_c)$, where
$\alpha = 0$ if $t_2 \geq t_1 + W_c + D_c$
$\alpha = 1$ otherwise.

Since W_c and D_c are functions of (i, s) and (i, j, s, l) respectively, *idletime* is a function of (i, j, s, l, t_1, t_2).

3.7 Problem Statement

Definition: Given a multiprocessor M and a DAG g to be executed on M, we say that an execution of g on M *does not violate* partition $\Pi = \{\tau_1, \tau_2, \ldots, \tau_n\}$, if and only if, each task τ_i of Π is executed in a single PE (i.e. all the nodes in τ_i are executed on the same PE).
Note that 2 nodes that are in different tasks may be executed on the same PE. However, 2 nodes that are in the same task have to be executed on the same PE.

Example: The trivial partition is not violated by any execution graph g on multiprocessor M.

Definition: A partition Π_1 is said to be *contained* in partition Π_2, if and only if, each task τ_i of Π_1 is included in some task τ_j of Π_2.[9] We write $\Pi_1 \subset \Pi_2$.[10] We also say that Π_2 is smaller than Π_1.

[9] τ_i could be the same as τ_j.

[10] This is not the same as the ususal set inclusion. We simply borrow this notation for convenience.

Example: The trivial partition is contained in any non-trivial partition

Corollary: For a multiprocessor M and DA g, if an execution of g on M does not violate some partition Π_1, then it does not violate any partition Π_2 which is contained in Π_1.

Corollary: For any 2 partitions Π_1 and Π_2, $\Pi_1 \subset \Pi_2 \Rightarrow \Pi_1$ has at least as many tasks as Π_2 does.

Proof: Straightforward: Use proof by contradiction: We assume that the statement of the corollary is not true, then using that we deduce a false statement.

Definition: Given a multiprocessor M and a DAG g to be executed on M, a *universal* partition is a non-trivial partition which is not violated by any execution of g on M, that leads to minimal parallel execution time for any number of PEs in M (including the infinite number).
Stated differently, a *universal* partition is a non-trivial partition Π_u which is contained in any optimal partition Π_{opt} (i.e. a partition that results into minimal parallel execution time) for any number of processors: $\Pi_u \subset \Pi_{opt}$.

The Partitioning Problem: Given a target multiprocessor M and a DAG g to be executed on M, the code partitioning problem consists of finding the smallest universal partition for g.

The Idea: The reason for choosing a universal partition is that when we start from such partition and start lumping tasks together, we are able to get to the optimal partition for the number of processors that are available, provided that our algorithm leads to optimal solution. This is true since a universal partition is contained in the optimal partition. When we start from the smallest universal partition, we save work since the smallest partition has the least amount of tasks. This lumping process is done in the scheduling phase as will be seen later[11].

[11]When 2 tasks are assigned to the same processor, we say that they are lumped together.

Remark (Intuitive fact): The number of tasks in the smallest universal partition should be greater than the number of PEs in the multiprocessor. If it is not, then the problem that we are trying to run does not have sufficient parallelism for all the PEs.

Theorem: Given a target multiprocessor M and a DAG g to be executed on M,
Let Π_∞ be The optimal partition for the following case:
We have a virtual DMM (DMM_v) satisfying the following 2 properties:

1. Infinite number of *virtual* PEs,

2. Communication overhead between the PEs is minimal (not zero)[12].

Π_∞ is the smallest universal partition.

Proof:

1. First, let's prove that Π_∞ is a universal partition.
   For any task τ_i in Π_∞, all nodes in τ_i have to be executed in the same processor to get optimal parallel execution time, given the best case situation of an infinite number of processors with minimum communication overhead. Therefore, in the realistic case of a finite number of processors with varying communication overhead, all the nodes in τ_i have to be executed in the same processor as well, in order to get optimal parallel execution time. Hence, any execution that leads to optimal performance does not violate Π_∞. As a consequence, Π_∞ is a universal partition.

[12]Here we assume that the communication overhead between any 2 virtual processors is minimal. In other words, we assume that the distance between any two processors is one hop (i.e. all processors are directly linked with one another). Also, we assume that the total communication load in the multiprocessor network is always negligible and does not affect the communication time between processors, and therefore we can ignore it. Hence in this case, the delay component D_c does not depend on the source processor, the destination processor, or the total communication load. D_c is then a function of the size of the message only. In addition, we assume that W_c and R_c for any virtual processor are equal to the minimum value of W_c and R_c respectively of all physical processors. Hence, W_c and R_c are also functions of the message size only.

2. Now we have to prove that Π_∞ is the smallest universal partition.
   Since Π_∞ is an optimal partition, then any universal partition Π_u is contained in it. Therefore Π_∞ is smaller than any universal partition. Since it is itself a universal partition, then it is the smallest universal partition.

3.7.1 Remarks

- Sarkar uses the same definition for the partitioning problem. However, our definition is much more formal.
- The code partitioning is usually done at compile-time. It is very unusual for parallel compilers to use dynamic code partitioning schemes. In this work, the compile-time method is used.

3.7.2 Why Π_∞?

Here, we give a more informal and easier to understand explanation for the choice of Π_∞.
Consider a task τ_i in Π_∞.
Let $\tau_i = \{a_1, a_2, \ldots, a_n\}$, where the a_j's are actors in the input program graph. Since under the ideal case of infinite number of PEs and minimum non-zero communication overhead actors $a_1, a_2, \ldots, a_n$ belong to the same task, then under the realistic case of finite number of PEs and actual communication overhead they have to belong to the same task as well. Hence all the actors that belong to the same task in Π_∞ belong to the same task in the optimal partition for the realistic case. Therefore, by doing some further merging of the tasks in Π_∞, we can obtain the optimal partition for the realistic case. If our scheduling algorithm is optimal, then the tasks that should be merged together will be assigned to the same PE. In our approach, Π_∞ is passed as the input to the scheduling phase, and we rely on the scheduling algorithm to assign the tasks that should be merged together to the same PE.

3.7.3 Overall Procedure

- Start with the *trivial partition.*
- Perform a sequence of partitioning refinements.
- At each step, the algorithm tries to improve on the previous partition by choosing a pair of tasks to be merged *using some heuristic*[13].
  We record the parallel execution time (PARTIME)[14] corresponding to the new partition.
- Stop when the singleton partition is reached.
- Choose partition with lowest PARTIME.

Remarks

- Most partitioning algorithms use the above overall approach.
- The main work of the algorithm is to find the appropriate tasks to be merged during each step.
  Therefore, we have to study very carefully the effects of task merging and understand its impact on CPL, available parallelism in the task graph, reduction in communication overhead, etc.
- We keep merging tasks until we reach the coarsest (*singleton*) partition.
  We will see in a later section that we need to keep merging tasks even if the merger results into a higher PARTIME. This is done so that we don't get caught at a *local minimum.*
- The parallelism granularity is determined by the size of the tasks in the partition that results from the partitioning analysis.

[13]Merging 2 tasks τ_i and τ_j means replacing them by a new task τ_k which contains all the actors in τ_i and τ_j: $\tau_k = \tau_i \cup \tau_j$.

[14]From now on, parallel execution time will be abbreviated to PARTIME.

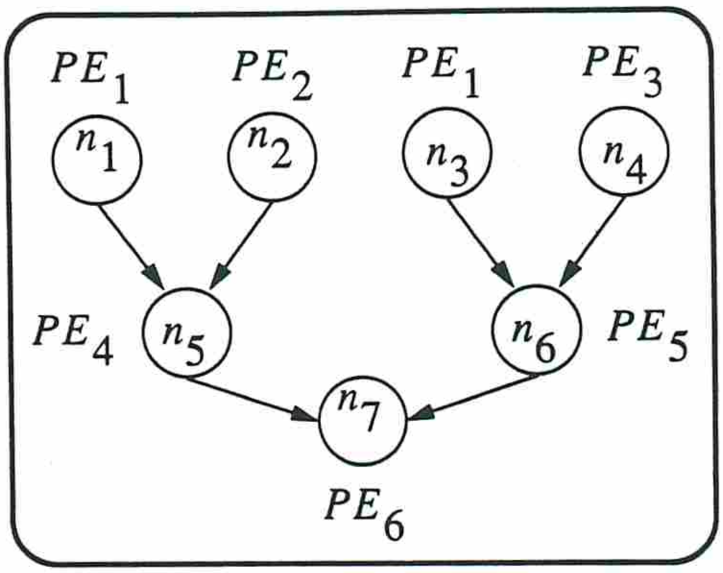


Figure 3.6: Two tasks mapped to the same PE

3.7.4 Estimating PARTIME

We make the following 2 assumptions:

1. At each step of the algorithm, each task of the current partition is mapped to a separate PE.
2. All the inputs of the program graph are ready before the program starts execution.

Therefore, we can estimate PARTIME to the CPL of the task graph of the current partition.

Remark

To see why the assumption stating that each task of the current partition is mapped to a separate PE is necessary, consider the task graph shown in figure 3.6. Nodes n_1 and n_3 are mapped to the same PE. When the program starts execution, these 2 nodes cannot start execution at the same time. Assume that n_1 starts execution first. Thus n_3 can start execution only when n_1 finishes. Therefore we have to add $comp(n_1)$ to the length of path (n_3, n_6, n_7) when we figure out PARTIME.

Notations and Definitions

Given a task graph.
Let n be a node in the graph.
Let $e = (n_1, n_2)$ be an edge in the graph connecting 2 nodes n_1 and n_2.
Let p be a path in the graph.
comp(n) := Cost of computation in node n.
data(e) := Amount of data communicated along edge e during 1 execution of the program.
data(n_i, n_j) := 0, if there is no edge (n_i, n_j).
comm(e) := Cost of communicating data on edge e during 1 execution of the program.
comp(e) := comp(n_1) + comp(n_2).
L(p) := Length of path p.
L(p) = $\sum_{n \in p} comp(n) + \sum_{e \in p} comm(e)$.
$L_b(p)$:= Length of path p before merger.
$L_a(p)$:= Length of path p after merger.
CPL_b and CPL_a are defined to be the CPL of the task graph before and after the merger respectively.

Consider two virtual PEs PE_1 and PE_2 belonging to DMM_v.
Let f_c be the cost to communicate a message from PE_1 to PE_2.
f_c is a function of the message size s only, because of the characteristics of DMM_v.
$f_c(s)$:= Cost to communicate a message of size s from PE_1 to PE_2.
$f_c(s)$ has 2 components: a message start-up component T_{start} and a delay component $delay(s)$.
$f_c(s) = T_{start} + delay(s)$.
$delay(s)$ is proportional to s.
Since the message start-up component is a constant and does not depend on the message size, then

$f_c(s_1 + s_2) \neq f_c(s_1) + f_c(s_2)$.

i.e. $f_c(s)$ is not proportional to s.

We have $delay(s_1 + s_2) = delay(s_1) + delay(s_2)$.

$f_c(s_1 + s_2) = T_{start} + delay(s_1 + s_2)$.

Thus, $f_c(s_1 + s_2) = T_{start} + delay(s_1) + delay(s_2)$.

Hence, $f_c(s_1 + s_2) = f_c(s_1) + delay(s_2)$.

We define $delay(s) := 0$, if $s = 0$.

$comm(e) = T_{start} + delay(data(e))$

3.7.5 Equivalent Problem Statement

The optimal partition is the one for which the corresponding task graph has the shortest CPL among all partitions. The task graph corresponding to the optimal partition is called *optimal task graph.*

3.7.6 Complexity

The partitioning problem is NP-complete [48]. Therefore, all we can do is find heuristics that give a performance as close to the optimal as possible.

3.7.7 The Algorithm

An informal description of the algorithm for the code partitioning follows:

```
ALGORITHM PartitionGraph
BEGIN
  Partition := Trivial_Partition  /* Start with the trivial partition.
  PARTIME := CPL of current task graph  /* Parallel Execution Time
                                        /* corresponding to current
                                        /* partition.
  best_partition := Partition  /* Best partition found so far.
  best_time := PARTIME  /* Best parallel execution time found so far.
  WHILE |partition| >= 2 DO
    BEGIN  /* Perform a merging iteration.
```

```
      Partition := Merge(H)  /* Choose a pair of tasks to be merged
                             /* using Heuristic H, and merge them.
                             /* Partition = partition after merger.
      PARTIME := CPL of new task graph
      IF (PARTIME < best_time)
        THEN
          BEGIN
            best_partition := Partition
            best_time := PARTIME
          END
    END
END
```

At the end of the execution of the algorithm, variable *best-partition* is the partition chosen by the algorithm, and variable *best-time* is its corresponding PARTIME.

3.7.8 Effect of Merging a Pair of Tasks

- If the tasks are independent ⇒
  . No reduction in communication cost.
  . Loss in parallelism.
- If the tasks are connected by an edge ⇒
  . Reduction in communication overhead[15].
  . Possible loss in parallelism.

Merging Tasks

Goal: Minimize CPL of task graph.

[15]Note that during the scheduling phase, these two tasks may be assigned to the same physical PE, and therefore no reduction in communication overhead is done as a result of merging them. However, since we assume that we have an infinite number of virtual PEs, and that each task is mapped to a different virtual PE, there is reduction in communication overhead as a result of the merger.

Rule: Merge only pairs of nodes connected by an edge, since there is no gain from merging nodes not connected by edges.

PARTIME := Parallel execution time of the program on the DMM.
PARTIME = $T_c + T_o$, where
T_c := Computation Time Component,
T_o := Overhead Component (communication overhead only, no scheduling overhead).
Trade-off between computation component and overhead component.
The more parallelism we exploit, the smaller T_c and the larger T_o will be, and vice versa.
CPL = $T_c + T_o$.
For DMMs, communication cost is quite high $\Rightarrow$ Try to reduce communication as much as possible.
In general, merge tasks $\Rightarrow$ T_c increases (loss in parallelism and more sequentialization) and T_o decreases (reduction in communication overhead).

3.8 Task Merging

2 possibilities:

1. *Explicit merging:*
   In this method, we keep track of how the task graph looks like during the execution of the partitioning algorithm. Each time tasks are merged, we update the task graph to reflect the new partition (i.e. after each merging step, we determine the task graph of the new partition).
   This is called explicit merging, because we explicitly reflect the task merging in the task graph.
   Using explicit merging helps the partitioning analysis. For instance, it allows us to compute the parallel execution time at each step of the merging process, which is simply the CPL of the corresponding task graph. Also, it allows us to keep track of the available parallelism between tasks and of the dependency relationship between tasks, which help with the choice of

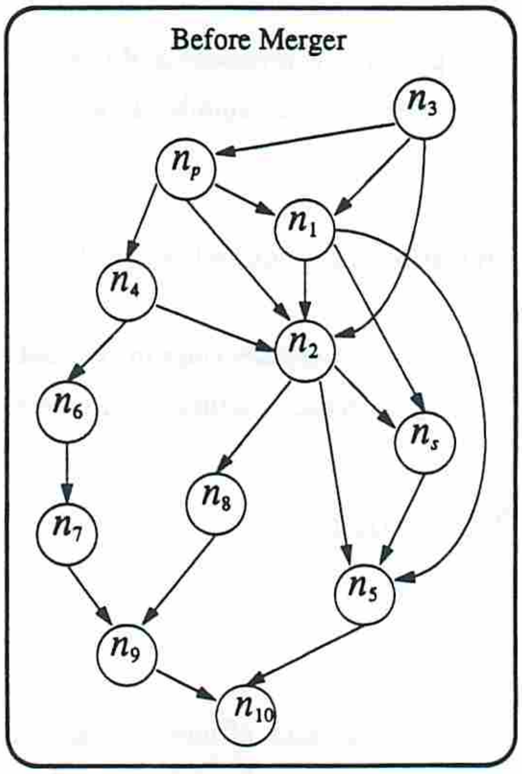


Figure 3.7: Explicit Task Merging

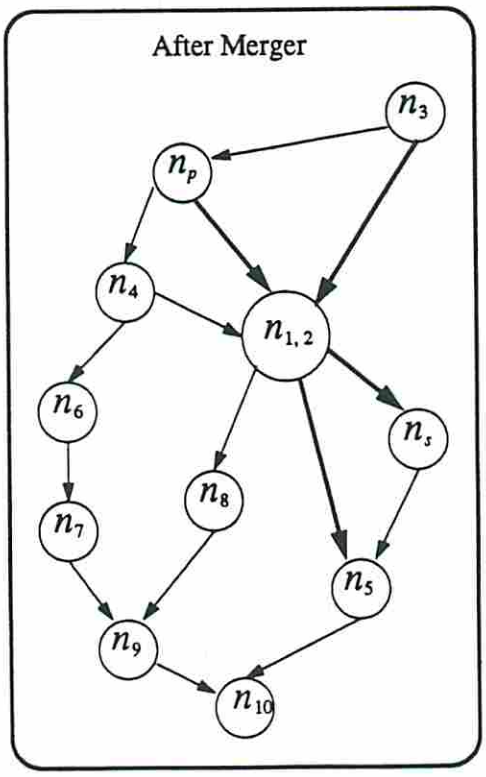


Figure 3.8: Explicit Task Merging

appropriate tasks to be merged.
Since at each merging step we have to update the task graph, we have to make sure that this updating of the graph has low time complexity.
For an example of this, refer to figures 3.7 and 3.8.
Figure 3.7 shows a task graph before merging nodes n_1 and n_2.
Figure 3.8 shows the task graph after the merger.

2. *Implicit merging:*
In this method, we don't keep up of how the task graph looks like during the execution of the partitioning algorithm.
The problem with this approach is that without knowing how the task graph looks like during each merging step, we cannot figure out the parallel execution time, and it is hard to figure out the parallelism available in the graph.

For our analysis, we use the explicit merging method.

Explicit Merging Procedure

In this section, we show how we update the task graph when tasks are merged.

For an example of this, refer to figures 3.7 and 3.8.
Figure 3.7 shows a task graph before merging nodes n_1 and n_2.
Figure 3.8 shows the graph after the merger.

Assumption: We assume that all messages inside a task which are destined to the same task are grouped together into a bigger message. There is no loss in doing so since no partial task execution is allowed because of the convexity constraint. Since tasks are generally small to medium grains, messages are never too big.

Initially, each actor in the input program graph is put in a separate task. Therefore, the task graph has one node for each actor in the program graph. The edges in the task graph are determined by the edges between the actors in the program graph.

Each time 2 nodes n_1 and n_2 in the task graph connected by an edge (n_1, n_2), are merged into a node $n_{1,2}$, we do the following:

- Nodes n_1 and n_2 are replaced by node $n_{1,2}$.
- Edge (n_1, n_2) is deleted.
- Any edge going from n_1 to any other node n_i $(i \neq 2)$ is replaced by an edge going from $n_{1,2}$ to n_i, and any edge going from n_i to n_1 is replaced by an edge going from n_i to $n_{1,2}$.
- Any edge going from n_2 to any other node n_i $(i \neq 1)$ is replaced by an edge going from $n_{1,2}$ to n_i, and any edge going from n_i to n_2 is replaced by an edge going from n_i to $n_{1,2}$.
- If there is an edge from n_1 to n_i and an edge from n_2 to n_i $(i \neq 1$ and $i \neq 2)$, then edges (n_1, n_i) and (n_2, n_i) are replaced by **one** edge $(n_{1,2}, n_i)$.
- If there is an edge from n_i to n_1 and an edge from n_i to n_2 $(i \neq 1$ and $i \neq 2)$, then edges (n_i, n_1) and (n_i, n_2) are replaced by **one** edge $(n_i, n_{1,2})$.

Merging an Edge in the Task Graph

Let $e = (n_1, n_2)$ be an edge in the task graph. Merging the edge e means merging tasks n_1 and n_2 together.

Time Complexity of Explicit Merging

Let N be the number of actors in the input program graph.
Initially, the task graph has N nodes as well.

Let's consider the cost of merging nodes n_1 and n_2.
Replacing these 2 nodes by node $n_{1,2}$ takes a constant amount of time.
Each of these 2 nodes is connected to at most $(N-1)$ other nodes. Therefore, the total cost to replace all edges during this merging step is $O(N)$. Hence, it costs at the most $O(N)$ time to explicitly merge nodes n_1 and n_2. Since there is a total of $(N-1)$ merging steps, then the total cost to do the explicit merging, counting all merging steps in the algorithm is $O(N^2)$.

Note that we over-estimated this time complexity because we assumed the worst case scenario. For real applications, the 2 nodes merged are not connected to all other nodes in the graph. on the average, the nodes in the task graph are connected to 3 or 4 nodes. Therefore, each merging step takes a constant amount of time. Hence the total cost to do the explicit merging, counting all merging steps in the algorithm is $O(N)$.

3.8.1 Updating Task Graph Weights as a Result of the Merger

Assume explicit merging.

Consider 2 nodes n_1 and n_2 in the DAG, connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$.

For an example of this, refer to figures 3.7 and 3.8.
Figure 3.7 shows a DAG g before merging nodes n_1 and n_2.
Figure 3.8 shows the graph g after the merger.
The thick edges represent edges that carry more data (i.e. the edge weights has increased).
The larger node represent a node that has more computations (i.e. the node weight has increased).

3.8.1.1 Node Weights

$comp(n_{1,2}) = comp(n_1) + comp(n_2)$.
Here we assume that all PEs are simple and are not capable of doing parallel computations[16].

3.8.1.2 Edge Weights

1. If an edge e' replaces **one** edge e:
   $data(e') = data(e)$.

[16]Even simple computations are done sequentially.

2. If an edge e' replaces **two** edges e_1 and e_2:
   $data(e') = data(e_1) + data(e_2)$.
   Since all messages inside a task that have the same destination task are combined together into a bigger message then:
   $comm(e') \neq comm(e_1) + comm(e_2)$.
   $comm(e') = T_{start} + delay(data(e_1)) + delay(data(e_2))$.
   $comm(e') = comm(e_1) + delay(data(e_2))$.
   $comm(e') = comm(e_2) + delay(data(e_1))$.

3.8.2 Creation of Cycles as a Result of Task Merging

The task graph should remain acyclic at all times, so that we guarantee that no deadlock situation occurs because of the convexity constraint rule. Hence the task graph should always be a DAG.
Initially the task graph is acyclic because we assume that the input program graph is acyclic. When we merge tasks, we have to make sure that no cycles are created as a result of the merger.

Consider 2 nodes n_1 and n_2 in the task graph, connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$.

Cycles could be introduced after the merger, because new edges are created. *All newly created edges are connected to the newly created node* $n_{1,2}$.

Theorem: Given an acyclic task graph, consider 2 nodes n_1 and n_2 in the graph, connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$. A cycle is created after the merger *if and only if* there exists a path from n_1 to n_2 other than (n_1, n_2) before the merger.

Proof

1. There exists a path from n_1 to n_2 other than (n_1, n_2) before the merger
   $\Longrightarrow$
   A cycle is created after the merger:
   The proof of this is quite obvious. Refer to figure 3.9.

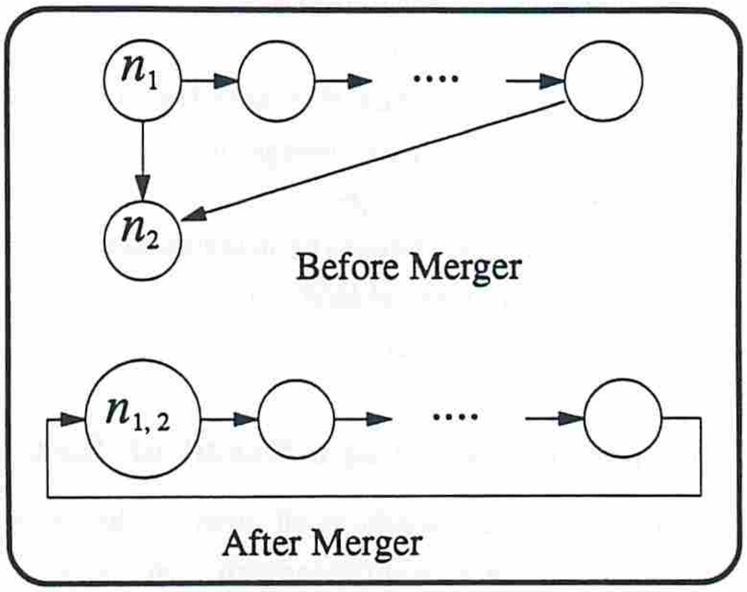


Figure 3.9: Cycle creation

2. A cycle is created after the merger

   $\Longrightarrow$

   There exists a path from n_1 to n_2 other than (n_1, n_2) before the merger:
   For a cycle to be introduced, a newly created edge has to have created it. Since any new edge is connected to $n_{1,2}$, then any created cycle contains the node $n_{1,2}$ (see figure 3.10-a).
   Before the merger, the portion of the graph in figure 3.10-a used to be the one shown in figure 3.10-b.
   We cannot have edges going from both nodes n_1 and n_2 to n_a, because that would create a cycle, and we no that the graph is acyclic before the merger. Also we cannot have edges going from n_b to both nodes n_1 and n_2, because that would also create a cycle.
   Assume that we have an edge from n_1 to n_a. Then we cannot have an edge from n_b to n_1 because that would create a cycle. Therefore, we can only have an edge from n_b to n_2 (see figure 3.10-c).
   Assume that we have an edge from n_2 to n_a. Then having an edge from n_b to n_1 or an edge from n_b to n_2 would create a cycle. Hence we cannot have

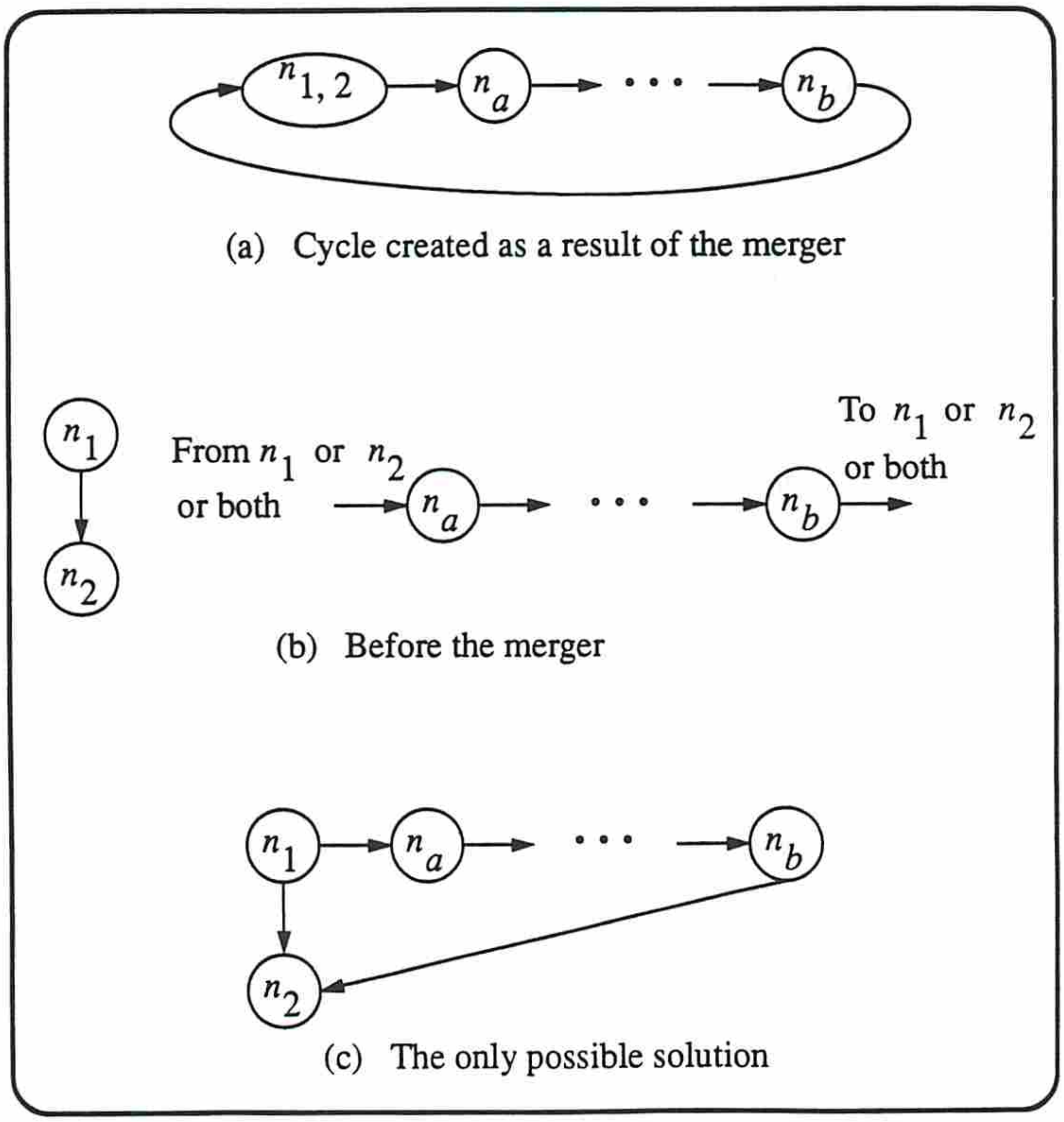

$n_{1,2}$
n_a
n_b
(a) Cycle created as a result of the merger
n_1
n_2
From n_1 or n_2 or both
To n_1 or n_2 or both
(b) Before the merger
(c) The only possible solution

Figure 3.10: Cycle creation

an edge from n_2 to n_a.

In conclusion, the only possibility is the one shown in figure 3.10-c.

A merging Rule

If there is a path from n_1 to n_2 in the task graph other than (n_1, n_2), then nodes n_1 and n_2 should not be merged, otherwise we create a cycle in the graph.
Keeping the task graph free of cycles guarantees that no deadlock situation occurs because of the convexity constraint.

Effect on Time Complexity

Because of the merging rule stated above, each time 2 tasks are chosen to be merged, we have to check for cycle creation before we do the merger. This check will add to the time complexity of the partitioning algorithm. In order to be efficient, we should try to find a way to do this check at a low time cost.

Effect on Efficiency of Partitioning Algorithm

Consider the 2 nodes n_i and n_j in the task graph, that are best candidates to be merged (i.e. their merger results into the best improvement in the partition). If their merger results into a cycle, then we cannot merge them, even though their merger results into the best improvement in the partition.

To get around this problem, we might consider merging nodes n_1 and n_2 and all the nodes that belong to all cycles created as a result of merging n_1 and n_2 together. This will guarantee that no cycles are created as a result of the merger. However, there is no guarantee that the partition obtained as a result of the merger is better than the one before the merger.

Chapter 4

Analysis of the Task Graph

4.1 Parallelism Loss Due to Task Merging

In this section, we study the effect of task merging on the available parallelism in the task graph.

Question: *Given a task graph, could there be any loss in parallelism when two nodes connected by an edge are merged?*

Answer: Yes

> It is quite obvious that some parallelism may be lost, even though these 2 nodes are connected by an edge, and therefore are dependent.

Proof: Assume that the answer is NO.
Then when we merge two nodes connected by an edge, there is reduction in communication overhead, and in addition no parallelism is lost. Therefore the optimal partition is obtained by merging any two nodes connected by an edge, which will result into a graph consisting of only ONE node (i.e. optimal partition consists of a single task)!!

Why?

Consider two nodes n_1 and n_2 connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$.

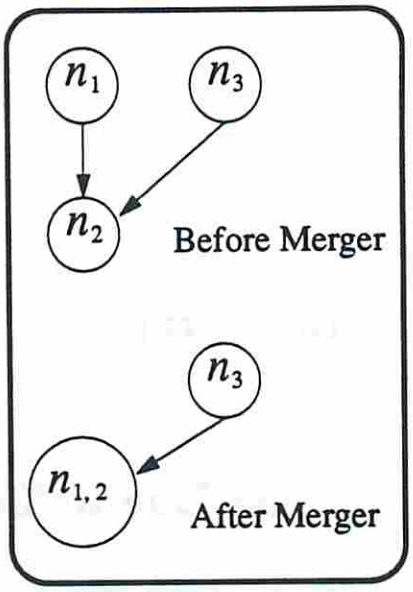


Figure 4.1: Parallelism Loss

Let's examine node n_1:
Let n_3 be a node in g such that n_1 and n_3 are independent, and n_2 and n_3 are dependent.
Then after the merger, n_1 and n_3 become dependent[1]. Thus n_1 and n_3 cannot be executed in parallel any longer. This represents a loss in parallelism with respect to n_1 in the task graph.
For an example of this, refer to figure 4.1.

4.1.1 Definitions

Parallel Set of a Node: Given a node n in a DAG g, the Parallel Set of n is $\text{ParSet}(n) := \{n' \in g \mid n \text{ and } n' \text{ are independent}\}$.
These are the nodes that can be executed in parallel[2] with n.
Given 2 nodes n_1 and n_2 in a DAG,

[1]To be more accurate, it is $n_{1,2}$ and n_3 which are dependent, since n_1 by itself no longer exists after the merger.
[2]n may not be executed in parallel with all nodes in ParSet(n) simultaneously, since ParSet(n) is not necessarily an independent set.

$\mathrm{ParSet}(n_1, n_2) := \mathrm{ParSet}(n_1) \cup \mathrm{ParSet}(n_2)$.

Dependent Set of a Node: Given a node n in a DAG g, the dependent set of n is

$\mathrm{DepSet}(n) := \{n' \in g \;/\; n \text{ and } n' \text{ are dependent}\}$

Given 2 nodes n_1 and n_2 in a DAG,

$\mathrm{DepSet}(n_1, n_2) := \mathrm{DepSet}(n_1) \cup \mathrm{DepSet}(n_2)$.

Remark: DepSet(n) is evaluated by finding all paths that pass through n.

Lemma: Consider two nodes n_1 and n_2 connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$,

$\mathrm{ParSet}(n_{1,2}) = \mathrm{ParSet}(n_1) \cap \mathrm{ParSet}(n_2)$

Remark: Also

$\mathrm{ParSet}(n_{1,2}) = \mathrm{ParSet}(n_1, n_2) - [\mathrm{ParSet}(n_1, n_2) \cap \mathrm{DepSet}(n_1, n_2)]$

Lemma: Consider two nodes n_1 and n_2 connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$,

$\mathrm{DepSet}(n_{1,2}) = \mathrm{DepSet}(n_1) \cup \mathrm{DepSet}(n_2)$

4.1.2 Parallelism with respect to a Node

Parallelism with respect to a Node: Given a node n in a DAG g, we define the Parallelism with respect to n to be $|ParSet(n)|$[3].

Parallelism Loss with respect to a Node: Given a node n in a DAG g, we say that there is parallelism loss with respect to n as a result of merging nodes *if and only if* the parallelism with respect to n after the merger is strictly smaller than the parallelism with respect to n before the merger.

[3]Given a set S, $|S|$ is the number of elements in S.

4.1.2.1 Condition for Parallelism Loss

Consider two nodes n_1 and n_2 in the task graph, connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$.

Definition: We say that some parallelism is lost in the task graph as a result of the merger *if and only if* some parallelism is lost with respect to n_1 or n_2.

Thus if we guarantee that no parallelism is lost with respect to n_1 **and** no parallelism is lost with respect to n_2, then we guarantee that no parallelism is lost in the task graph as a result of the merger.

$ParSet(n_1)$ represents all nodes which can be executed in parallel with n_1 before the merger.
Any node belonging to $DepSet(n_2)$ cannot be executed in parallel with n_1 after the merger.
Therefore, the set $\mathrm{ParSet}(n_1) \cap \mathrm{DepSet}(n_2)$ represents all nodes which could execute in parallel with n_1 before the merger, and no longer can be executed in parallel with n_1 after the merger.

The same analysis can be applied to node n_2.

Hence:

1. Some parallelism will be lost with respect to n_1 as a result of the merger *if and only if*
   $\mathrm{ParSet}(n_1) \cap \mathrm{DepSet}(n_2) \neq \emptyset$.
   An equivalent condition is:
   $ParSet(n_{1,2}) \neq ParSet(n_1)$.

2. Some parallelism will be lost with respect to n_2 as a result of the merger *if and only if*
   $\mathrm{ParSet}(n_2) \cap \mathrm{DepSet}(n_1) \neq \emptyset$.
   An equivalent condition is:
   $ParSet(n_{1,2}) \neq ParSet(n_2)$.

4.1.2.2 Amount of Parallelism Lost

Consider 2 nodes n_1 and n_2 in the task graph, connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$.

1. The amount of parallelism lost (if any) with respect to n_1 as a result of the merger is defined to be
   $|ParSet(n_1) \cap DepSet(n_2)|$.
   This can also be expressed as:
   $|ParSet(n_1)| - |ParSet(n_{1,2})|$.

2. The amount of parallelism lost (if any) with respect to n_2 as a result of the merger is defined to be
   $|ParSet(n_2) \cap DepSet(n_1)|$.
   This can also be expressed as:
   $|ParSet(n_2)| - |ParSet(n_{1,2})|$.

Definition: The amount of parallelism lost in the task graph as a result of the merger is defined to be the sum of the amount of parallelism lost with respect to n_1 and the amount of parallelism lost with respect to n_2.

4.1.2.3 Remark 1

Consider the example in figure 4.2.
ParSet(n_2) = $\{n_3, n_4\}$.
DepSet(n_1) = $\{n_4, n_2\}$.
ParSet$(n_2) \cap$ DepSet(n_1) = $\{n_4\} \neq \emptyset$.
Hence there is parallelism loss with respect to n_2.
Note that ParSet(n_2) is not an independent set. n_3 and n_4 are dependent, and therefore n_2 cannot execute in parallel with these 2 nodes simultaneously. Hence, if only one of these 2 nodes becomes dependent with n_2 after the merger, n_2 can still execute in parallel with the other node. Nevertheless, the CPL can still increase here. However, if both n_3 and n_4 become dependent with n_2 after the merger, then node n_2 will not be able to execute in parallel with any of these 2

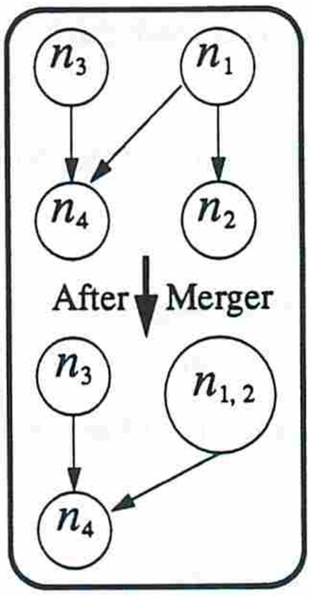


Figure 4.2: Parallelism Loss

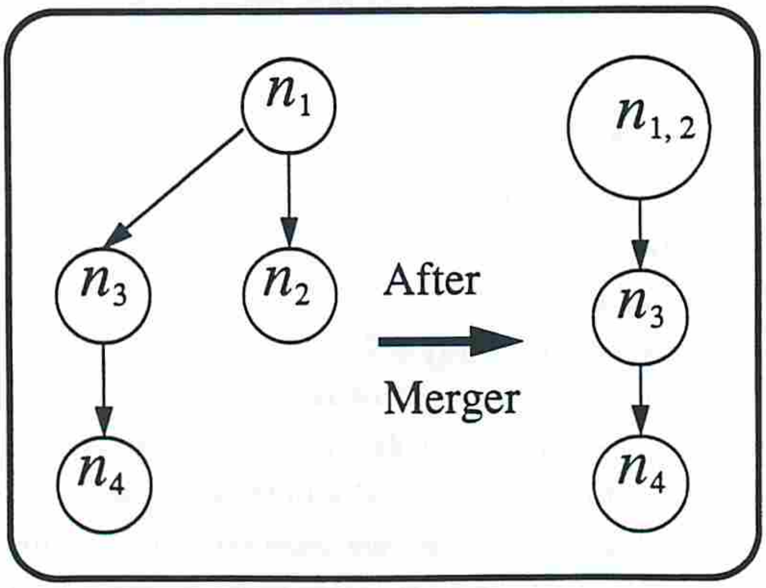


Figure 4.3: Parallelism Loss

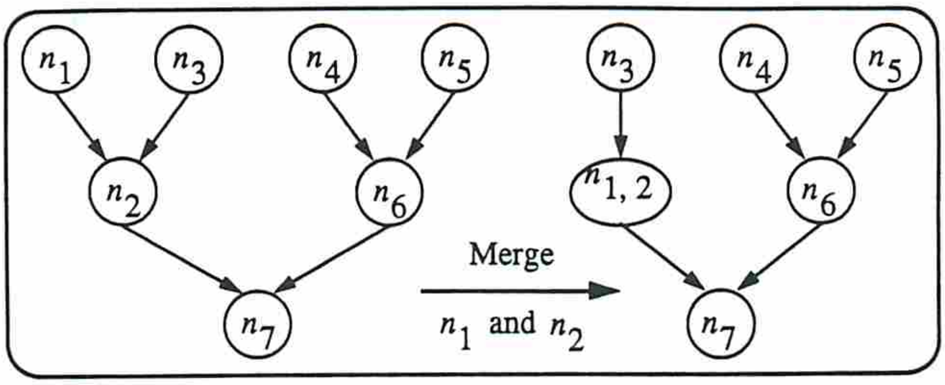


Figure 4.4: Example 1

nodes. For an example of this refer to figure 4.3.

In figure 4.3,

$\text{ParSet}(n_2) = \{n_3, n_4\}$.

$\text{DepSet}(n_1) = \{n_3, n_4, n_2\}$.

$\text{ParSet}(n_2) \cap \text{DepSet}(n_1) = \{n_3, n_4\} \neq \emptyset$.

4.1.2.4 Remark 2

Consider the following 2 examples.

Example 1

Given the task graph in figure 4.4.

Before the merger:

$\text{ParSet}(n_1) = \{n_3, n_4, n_5, n_6\}$.

n_1, n_3, n_4 and n_5 can execute in parallel.

After the merger:

$\text{ParSet}(n_{1,2}) = \{n_4, n_5, n_6\}$.

We have parallelism loss with respect to n_1 (one node: n_3 lost). However, n_3, n_4 and n_5 can still execute in parallel. In other words, only one node is lost for parallelism (node n_1).

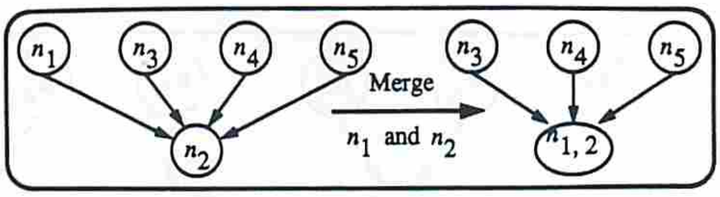


Figure 4.5: Example 2

Example 2

Given the task graph in figure 4.5.

Before the merger:

ParSet(n_1) = $\{n_3, n_4, n_5\}$.

n_1, n_3, n_4 and n_5 can execute in parallel.

After the merger:

ParSet($n_{1,2}$) = $\emptyset$.

All the parallelism with respect to n_1 is lost (3 nodes). However, n_3, n_4 and n_5 can still execute in parallel. In other words, only one node is lost for parallelism (node n_1).

Let $S = ParSet(n_1) \cap DepSet(n_2)$. For all nodes $n \in S$, there exists at least one path p which goes through n and n_2 but not n_1 (before the merger). Therefore, p increases in length after the merger (see section 4.2). The more nodes S has the more execution paths are most likely to increase after the merger. Hence defining the amount of parallelism with respect to n_1 as $|S|$ makes sense. Note that in general the number of execution paths that increase as a result of merging tasks is different from $|S|$.

4.1.3 Defining Parallelism

It is very hard to define the parallelism available in a task graph formally and precisely. All we can do is give an approximative definition. The definition should depend on what we need it for and how we are going to use it. For instance in our case, our measure of the performance of the partitioning algorithm is the CPL.

We will see later on in the analysis that our definition of parallelism is tightly coupled with the CPL of the task graph, and that the way the CPL is affected by the merger of tasks is related to the loss of parallelism (the way we defined it) in the task graph.

Some other possible definitions of the parallelism in a task graph follow. Note that we assume that all nodes have the same weight and all edges have the same weight.
Total Parallelism = $\sum_{n \in P_{crit}}(1 + MaxPar(n))$.
Maximum Parallelism = $Max_{n \in P_{crit}}\{1 + MaxPar(n)\}$.

4.1.4 Usable Parallelism

Degree of Parallelism: Given a DAG g and a set S of nodes belonging to g, the degree of parallelism in S is $|P|$, where[4] P is the largest parallel set of S.

Usable Parallelism: Given a DAG g and a node n in g, the Usable Parallelism with respect to n is MaxPar(n), defined to be the degree of parallelism in ParSet(n).

Lemma: Given a DAG g and a node n in g, the Usable parallelism with respect to n is the maximum number of nodes in g that can be executed in parallel *simultaneously* with node n.

4.1.4.1 Condition for Usable Parallelism Loss

Consider 2 nodes n_1 and n_2 in the task graph, connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$.

1. There is usable parallelism loss with respect to n_1 as a result of the merger *if and only if*
   MaxPar(n_1) > MaxPar($n_{1,2}$).

[4] $|P|$ is the number of elements in the set P.

2. There is usable parallelism loss with respect to n_2 as a result of the merger *if and only if*
   $\text{MaxPar}(n_2) > \text{MaxPar}(n_{1,2})$.

4.1.4.2 Amount of usable Parallelism Lost

Consider 2 nodes n_1 and n_2 in the task graph, connected by an edge (n_1, n_2) and merged into a node $n_{1,2}$.

1. The amount of usable parallelism lost (if any) with respect to n_1 as a result of the merger is defined to be
   $\text{MaxPar}(n_1) - \text{MaxPar}(n_{1,2})$.

2. The amount of usable parallelism lost (if any) with respect to n_2 as a result of the merger is defined to be
   $\text{MaxPar}(n_2) - \text{MaxPar}(n_{1,2})$.

4.1.4.3 Another condition for usable Parallelism Loss

In what follows, we assume that two nodes n_1 and n_2 in the task graph, connected by an edge (n_1, n_2) are merged into a node $n_{1,2}$.

Let's examine the parallelism lost with respect to n_1.
Let $S = \text{ParSet}(n_1) \cap \text{DepSet}(n_2)$.
S represents all the nodes which could be executed in parallel with n_1 before the merger, and no longer can execute in parallel with n_1 after the merger.
However, $\text{ParSet}(n_1)$ is not necessarily an independent set, and therefore it is not always the case that n_1 can execute with all nodes in $\text{ParSet}(n_1)$ simultaneously.
Let's assume that $\text{ParSet}(n_1)$ is not an independent set.
Hence, there exists at least one dependent set $S' \subset \text{ParSet}(n_1)$.
Clearly, no 2 nodes belonging to S' can execute in parallel.
Hence, n_1 can execute in parallel with **only one** node **at a time** from S'.
Therefore, if after the merger some nodes in S' become dependent with n_1 because they used to belong to $\text{DepSet}(n_2)$ before the merger, no parallelism will be lost with respect to n_1 because of that, provided that at least one node in S' remains independent with n_1 after the merger.

The Condition

Some usable parallelism will be lost in the task graph as a result of the merger *if and only if* any of the 2 following conditions is true:

1. Some usable parallelism with respect to n_1 is lost.

2. Some usable parallelism with respect to n_2 is lost.

Usable Parallelism Lost With Respect to n_1

Case 1 ParSet(n_1) is an independent set[5].

In this case, some parallelism will be lost with respect to n_1 *if and only if* ParSet(n_1) $\cap$ DepSet(n_2) $\neq \emptyset$.

Case 2 ParSet(n_1) is not an independent set.

Therefore, there is at least one dependent set $S' \subset$ ParSet(n_1).
Without any loss of generality, let's assume that ParSet(n_1) has k anti-parallel sets: $S_1, S_2, \ldots, S_k$.
Let $S_u = S_1 \cup S_2 \cup \ldots \cup S_k$.
Let $S =$ ParSet(n_1) $\cap$ DepSet(n_2).
In this case, some parallelism will be lost with respect to n_1 *if and only if* the following 2 conditions C_1 and C_2 are satisfied:

1. C_1: $S \neq \emptyset$.

2. $C_2 = C_{2,1}$ OR $C_{2,2}$.
   $C_{2,1}$: $S \not\subset S_u$
   (i.e. there exists at least one node $n \in S$ such that $n \notin S_u \Leftrightarrow n$ does not belong to any of the sets S_i, $1 \leq i \leq k \Leftrightarrow n$ and any other node in ParSet(n_1) are independent.)
   $C_{2,2}$[6]: $S \subset S_u$ and at least one of the sets $S_i \subset S$, $1 \leq i \leq k$.

[5]There is no dependent set $S' \subset$ ParSet(n_1).

[6]Here we are assuming that all sets S_i's are disjoint, and that no two nodes belonging to different anti-parallel sets of ParSet(n_1) can be dependent.

Usable Parallelism Lost With Respect to n_2

The exact same analysis that applies to n_1 applies to n_2 as well.

Problem With Above Condition

As was mentioned earlier, for condition $C_{2,2}$, we assume that all sets S_i's are disjoint, and that no two nodes belonging to different anti-parallel sets of ParSet(n_1) can be dependent. Clearly, in general this might not be true.

4.1.5 Relationship between Parallelism and Usable Parallelism

Lemma: Given a DAG g and a node n in g,
PAR(n) $\geq$ MaxPar(n)
i.e. the parallelism w/ respect to n is greater or equal than the usable parallelism w/ respect to n.

Lemma: Given a DAG g and a node n in g,
There is usable parallelism loss with respect to $n \Longrightarrow$
There is parallelism loss with respect to n.
There is parallelism loss with respect to $n \not\Longleftarrow$
There is usable parallelism loss with respect to n.

There is no parallelism loss with respect to $n \Longrightarrow$
There is no usable parallelism loss with respect to n.

There is no usable parallelism loss with respect to $n \not\Longrightarrow$
There is no parallelism loss with respect to n.

Lemma: Given a DAG g and a node n in g, if $ParSet(n)$ is an independent set, then
PAR(n) = MaxPar(n).

Lemma: Given a DAG g and a node n in g, if $ParSet(n)$ is an independent set, then:

There is parallelism loss with respect to $n \iff$
there is usable parallelism loss with respect to n.

4.1.6 Upper Bound on Degree of Parallelism

Consider a DAG g and a set S of nodes belonging to g.
We find the smallest k such that
$S = S_1 \cup S_2 \cup \ldots \cup S_k \cup S_{ind}$,
where each S_i is an anti-parallel set, and S_{ind} is a set of zero or more nodes that don't belong to any dependent set.
$k = 0$ represents the case for which S doesn't have any dependent sets (i.e. S is an independent set).
Any parallel set of S contains zero or one element from each set S_i, plus all elements in S_{ind}.
Therefore, *the degree of parallelism in S is* $\leq k + |S_{ind}|$.
Consider the case where S can be written as
$S = S_1 \cup S_2 \cup \ldots \cup S_k \cup S_{ind}$,
where each S_i is an anti-parallel set, and S_{ind} is a set of zero or more nodes that don't belong to any dependent set, and all S_i's are disjoint, and no 2 nodes which belong to different anti-parallel sets (from the above listed anti-parallel sets) are dependent.
In this case, *the degree of parallelism* $= k + |S_{ind}|$.

4.1.7 Theorem

Let g be a task graph. Let n_1 and n_2 be 2 nodes in g connected by an edge $e = (n_1, n_2)$.
Let $S_1 := DepSet(n_1) - \{n_2\}$.
Let $S_2 := DepSet(n_2) - \{n_1\}$.
$ParSet(n_1) \cap DepSet(n_2) = \emptyset$ AND $ParSet(n_2) \cap DepSet(n_1) = \emptyset$
$\iff$
$ParSet(n_1) = ParSet(n_2)$

$\iff$

$S_1 = S_2$.

Proof

1. Assume that: $ParSet(n_1) \cap DepSet(n_2) = \emptyset$ AND $ParSet(n_2) \cap DepSet(n_1) = \emptyset$.

   (a) $ParSet(n_1) \cap DepSet(n_2) = \emptyset$:
   $DepSet(n_2) = \{n_1\} \cup S_2$.
   $\forall n \in S_2, n \notin ParSet(n_1) \Rightarrow$
   $\forall n \in S_2, n \in DepSet(n_1)$ (since $n \neq n_1$) $\Rightarrow$
   $\forall n \in S_2, n \in S_1$ (since $n \neq n_2$) $\Rightarrow$
   $S_2 \subset S_1$ (1)
   $\forall n \in ParSet(n_1), n \notin DepSet(n_2) \Rightarrow$
   $\forall n \in ParSet(n_1), n \in ParSet(n_2)$ (since $n \neq n_2$) $\Rightarrow$
   $ParSet(n_1) \subset ParSet(n_2)$ (2)

   (b) $ParSet(n_2) \cap DepSet(n_1) = \emptyset$:
   $DepSet(n_1) = \{n_2\} \cup S_1$.
   $\forall n \in S_1, n \notin ParSet(n_2) \Rightarrow$
   $\forall n \in S_1, n \in DepSet(n_2)$ (since $n \neq n_2$) $\Rightarrow$
   $\forall n \in S_1, n \in S_2$ (since $n \neq n_1$) $\Rightarrow$
   $S_1 \subset S_2$ (3)
   $\forall n \in ParSet(n_2), n \notin DepSet(n_1) \Rightarrow$
   $\forall n \in ParSet(n_2), n \in ParSet(n_1)$ (since $n \neq n_1$) $\Rightarrow$
   $ParSet(n_2) \subset ParSet(n_1)$ (4)

   (1) AND (3) $\Rightarrow S_1 = S_2$.
   (2) AND (4) $\Rightarrow ParSet(n_1) = ParSet(n_2)$.

2. Assume that:
   $ParSet(n_1) = ParSet(n_2)$.

$\forall n \in DepSet(n_2),\ n \notin ParSet(n_2) \Rightarrow$
$\forall n \in DepSet(n_2),\ n \notin ParSet(n_1) \Rightarrow$
$ParSet(n_1) \cap DepSet(n_2) = \emptyset$ (5)
$\forall n \in DepSet(n_1),\ n \notin ParSet(n_1) \Rightarrow$
$\forall n \in DepSet(n_1),\ n \notin ParSet(n_2) \Rightarrow$
$ParSet(n_2) \cap DepSet(n_1) = \emptyset$ (6)
(5) AND (6) $\Rightarrow S_1 = S_2$.

3. Assume that:
   $S_1 = S_2$.
   $\forall n \in ParSet(n_1),\ n \notin DepSet(n_1) \Rightarrow$
   $\forall n \in ParSet(n_1),\ n \notin S_1 \Rightarrow$
   $\forall n \in ParSet(n_1),\ n \notin S_2 \Rightarrow$
   $\forall n \in ParSet(n_1),\ n \notin DepSet(n_2)$ (since $n \neq n_1$) $\Rightarrow$
   $\forall n \in ParSet(n_1),\ n \in ParSet(n_2)$ (since $n \neq n_2$) $\Rightarrow$
   $ParSet(n_1) \subset ParSet(n_2)$ (7)
   $\forall n \in ParSet(n_2),\ n \notin DepSet(n_2) \Rightarrow$
   $\forall n \in ParSet(n_2),\ n \notin S_2 \Rightarrow$
   $\forall n \in ParSet(n_2),\ n \notin S_1 \Rightarrow$
   $\forall n \in ParSet(n_2),\ n \notin DepSet(n_1)$ (since $n \neq n_2$) $\Rightarrow$
   $\forall n \in ParSet(n_2),\ n \in ParSet(n_1)$ (since $n \neq n_1$) $\Rightarrow$
   $ParSet(n_2) \subset ParSet(n_1)$ (8)
   (7) AND (8) $\Rightarrow ParSet(n_1) = ParSet(n_2)$.

Corollary

Let g be a task graph. Let n_1 and n_2 be 2 nodes in g connected by an edge $e = (n_1, n_2)$.
No parallelism is lost in the task graph as a result of the merger
$\Longleftrightarrow$
$ParSet(n_1) = ParSet(n_2)$.

Proof

No parallelism is lost as a result of the merger *if and only if* no parallelism is lost with respect to n_1 and no parallelism is lost with respect to n_2.
This is true *if and only if*
$ParSet(n_1) \cap DepSet(n_2) = \emptyset$ AND $ParSet(n_2) \cap DepSet(n_1) = \emptyset$.
From the above theorem, that is true *if and only if*
$ParSet(n_1) = ParSet(n_2)$.

4.2 Effect of Task Merging on CPL

4.2.1 Problem Statement

In all what follows, we assume that 2 nodes n_1 and n_2 in the task graph connected by an edge $e = (n_1, n_2)$, are merged into a node $n_{1,2}$.
Let $p_c = P_{crit}$ of task graph before the merger.
$l_b :=$ length of P_{crit} of task graph before the merger.
$l_b = L_b(p_c)$.
$CPL_b := l_b$.
$l_a :=$ length of P_{crit} of task graph after the merger.

4.2.2 Effect on Path Length

Let p be any path in the task graph.
We have 3 possibilities:

1. **None** of the two nodes merged belongs to p.

2. **Only one** of the two nodes merged belongs to p.

3. **Both** nodes merged belong to p: $\Rightarrow e \in p$.
   To see why this is true, assume that $e \notin p$.
   $\Rightarrow$ there are two possibilities:

(a) There is a path from n_1 to n_2 other than (n_1, n_2).
$\Rightarrow$ After a merger, a cycle will be created.
Therefore n_1 and n_2 cannot be merged together.

(b) There is a path from n_2 to n_1.
$\Rightarrow$ There is a cycle before the merger, because of edge (n_1, n_2), which is not possible since we have a DAG.

The length of p is affected by the merger *if and only if* at least one of the following conditions is true:

1. A node in p is replaced by another node that has more computations (this is the case when only one of the 2 merged nodes belongs to p).
Assuming that $n_1 \in p$,
$L(p)$ is increased by $comp(n_2)$.

2. An edge in p is deleted (this is the case when $e \in p$).
$L(p)$ is reduced by $comm(e)$.

3. An edge in p is replaced by another edge which carries more data (this is the case when **two** edges e_1 and e_2 are replaced by **one** edge e', and either e_1 or e_2 belongs to p).
Assume $e_1 \in p$, then
$L(p)$ is increased by $delay(data(e_2))$.

There are 3 cases:

Case 1 **None** of the two nodes merged belongs to p:
$L_a(p) = L_b(p)$.

Case 2 **Only one** of the two nodes merged (say it is n_1) belongs to p:
Let n_p be the predecessor of n_1 in p (if any).
Let n_s be the successor of n_1 in p (if any).
$L_a(p) = L_b(p) + comp(n_2) + delay(data(n_p, n_2)) + delay(data(n_2, n_s))$.
Note that if (n_p, n_2) and (n_2, n_s) don't exist (or if n_p and n_s don't exist), then

$L_a(p) = L_b(p) + comp(n_2)$.
This increase in length of p represents a loss in parallelism and increase in sequentialization by the amount $comp(n_2)+delay(data(n_p, n_2))+delay(data(n_2, n_s))$ relative to path p.
The terms involving the *delay* function are due to the fact that some inter-PE communication has to be sequentialized are a result of the merger. For instance, the increase by the amount $delay(data(n_p, n_2))$ is due to the fact that before the merger, n_p used to send the data on edges (n_p, n_1) and (n_p, n_2) to separate virtual PEs in parallel. After the merger, the data on these 2 edges is combined and sent to the same virtual PE. Clearly this takes more time.
In conclusion, we could have an increase in the CPL, and as a consequence the parallel execution time could increase.
For an example of this, refer to figures 3.7 and 3.8.
Figure 3.7 shows a task graph before merging nodes n_1 and n_2.
Figure 3.8 shows the graph after the merger.
Consider path $p_1 = (n_3, n_p, n_1, n_s, n_5, n_{10})$ in figure 3.7. After the merger, $p_1 = (n_3, n_p, n_{1,2}, n_s, n_5, n_{10})$.

Case 3 Both nodes merged belong to p:
Let n_p be the predecessor of n_1 in p (if any).
Let n_s be the successor of n_2 in p (if any).
$L_a(p) = L_b(p) - comm(e) + delay(data(n_p, n_2)) + delay(data(n_1, n_s))$.
Note that if (n_p, n_2) and (n_1, n_s) don't exist (or if n_p and n_s don't exist), then
$L_a(p) = L_b(p) - comm(e)$.
This decrease in the length of p represents a reduction in communication overhead by the amount $x = comm(e)-delay(data(n_p, n_2))-delay(data(n_1, n_s))$ relative to path p (assuming that $x > 0$, which is true for most cases).
Again, the terms involving the *delay* function are due to the fact that some inter-PE communication has to be sequentialized are a result of the merger.
For an example of this, refer to figures 3.7 and 3.8.
Figure 3.7 shows a task graph before merging nodes n_1 and n_2.

Figure 3.8 shows the graph after the merger.

Consider path $p_2 = (n_3, n_p, n_1, n_2, n_s, n_5, n_{10})$ in figure 3.7. After the merger, $p_2 = (n_3, n_p, n_{1,2}, n_s, n_5, n_{10})$.

4.2.3 Merging an Edge Belonging to the Critical Path

In what follows, we omit the terms involving the function *delay* in the expressions giving the length of a path after merging two nodes, in terms of its length before the merger.

Let's assume that $e \in P_{crit}$ of task graph.
Thus, $L_a(p_c) = l_b - comm(e)$.

4.2.3.1 Effect on Execution Paths

Clearly for any execution path p in the task graph, $L_b(p) \leq l_b$, since l_b is the CPL before the merger.

1. Any execution path p that doesn't go through any of the 2 nodes merged:
   $L_a(p) = L_b(p)$.
2. Any execution path p that goes through n_1 and not n_2:
   $L_a(p) = L_b(p) + comp(n_2)$.
3. Any execution path p that goes through n_2 and not n_1:
   $L_a(p) = L_b(p) + comp(n_1)$.
4. Any execution path p that goes through edge e:
   $L_a(p) = L_b(p) - comm(e)$.

4.2.3.2 Effect on Critical Path

Case 1 There is no execution path that goes through only one of the 2 nodes merged:
Thus for any execution path p, $L_a(p) \leq L_b(p)$.
Also we know that $L_b(p) \leq l_b$. Therefore, $L_a(p) \leq l_b$.
Hence, *CPL will either decrease or remain unchanged as a result of the*

merger.

P_{crit} could change as a result of the merger. We have 2 cases:

1. If all execution paths p go through e:
   $L_a(p) = L_b(p) - comm(e)$.
   In this case, all execution paths including p_c will be reduced in length by the same amount.
   Hence, P_{crit} will not change and CPL decreases by $comm(e)$ as a result of the merger.
2. At least one execution path doesn't go through e:
   Let $p_1, p_2, \ldots, p_k$ be such execution paths, where $k \geq 1$.
   $L_a(p_i) = L_b(p_i)$, $1 \leq i \leq k$.
   There are 2 cases:
   (a) If at least one of the p_i's is such that
       $L_b(p_i) = l_b$:
       P_{crit} will change and CPL will remain unchanged.
   (b) If $L_b(p_i) < l_b$, $1 \leq i \leq k$:
       CPL will decrease.
       P_{crit} could change. There are 2 possibilities:
       i. $L_b(p_i) \leq L_a(p_c)$, $1 \leq i \leq k$:
          CPL will decrease by $comm(e)$.
          P_{crit} will not change.
       ii. At least one of the p_i's is such that $L_b(p_i) > L_a(p_c)$:
          P_{crit} will change.
          CPL will decrease by an amount smaller than $comm(e)$.
          Let p_m be the p_i such that $L_b(p_m)$ is the largest among all p_i's.
          After the merger,
          $P_{crit} = p_m$ and $CPL = L_b(p_m)$.
          CPL will decrease by $l_b - L_b(p_m)$.

Case 2 There is at least one execution path p that goes through only one of the 2 nodes merged:

P_{crit} could change as a result of the merger, and CPL could increase, since

the length of p increases after the merger.
Let $p_1, p_2, \ldots, p_k$ be all execution paths that go through only one of the 2 nodes merged, where $k \geq 1$.
Let $n_{i,1}$ and $n_{i,2}$ be the two nodes merged, and let $n_{i,1}$ be the node that belongs to p_i, and let $n_{i,2}$ be the other node[7], $1 \leq i \leq k$.
$L_a(p_i) = L_b(p_i) + comp(n_{i,2})$, $1 \leq i \leq k$.
We know that $l_b \geq L_b(p_i)$, $1 \leq i \leq k$, since l_b is the CPL before the merger.

There are so many possibilities, depending on the length of the execution paths before the merger, l_b, the value of $comp(n_{i,2})$, the value of $comm(e)$, etc.

Since we already studied the case where no execution path goes through only one of the 2 nodes merged, **let's assume that all execution paths (p_c excluded) go through only one of the 2 nodes merged**. This will simplify our analysis.
In this case, the execution paths are $p_1, p_2, \ldots, p_k$ and p_c.
There are 2 possible situations:

1. If $L_a(p_i) \leq L_a(p_c)$, $1 \leq i \leq k$:
   P_{crit} will not change.
   CPL will decrease by $comm(e)$.
2. If there is at least one execution path p such that $L_a(p) > L_a(p_c)$:
   P_{crit} will change, but CPL does not necessarily increase. We have 2 cases:
   (a) If $L_a(p_i) \leq l_b$, $1 \leq i \leq k$:
      i. If at least one of the p_i's is such that $L_a(p_i) = l_b$:
         Let p_o be this p_i.
         After the merger,
         $P_{crit} = p_o$.
         CPL remains unchanged.

[7]If p_i goes through n_1 then $n_{i,1}$ is n_1 and $n_{i,2}$ is n_2. If p_i goes through n_2 then $n_{i,1}$ is n_2 and $n_{i,2}$ is n_1.

ii. If $L_a(p_i) < l_b$, $1 \leq i \leq k$:
CPL will decrease by an amount less than $comm(e)$.
Let p_m be the p_i such that $L_a(p_m)$ is the largest among all p_i's.
After the merger,
$P_{crit} = p_m$ and $CPL = L_a(p_m)$.
CPL will decrease by $l_b - L_b(p_m) - comp(n_{m,2})$.

(b) If there is at least one execution path p such that $L_a(p) > l_b$:
CPL will increase.
Let p_m be the p_i such that $L_a(p_m)$ is the largest among all p_i's.
After the merger,
$P_{crit} = p_m$ and $CPL = L_a(p_m)$.
CPL will increase by $L_b(p_m) + comp(n_{m,2}) - l_b$.

4.2.3.3 Conclusion

- Merging an edge that belongs to P_{crit} of task graph does not guarantee a decrease in CPL.

- The maximum decrease in CPL is $comm(e)$.

- Merging an edge that belongs to all execution paths guarantees the maximum decrease in CPL.

- If none of the execution paths go through only one of the 2 nodes merged, then CPL will either decrease or remain unchanged.
  Also the maximum decrease in CPL could be achieved here.

- If at least one execution path goes through only one of the 2 nodes merged, then CPL will either increase, remain unchanged or decrease.
  Also the maximum decrease in CPL could be achieved here.

4.2.4 Merging an Edge Not Belonging to the Critical Path

In what follows, we omit the terms involving the function *delay* in the expressions giving the length of a path after merging two nodes, in terms of its length before the merger.

Let's assume that $e \notin P_{crit}$.

Clearly, e belongs to at least one execution path[8].

There are 2 possible cases:

Case 1 P_{crit} goes through only 1 of the 2 nodes merged:

Let n_{in} be the node merged which belongs to p_c, and let n_{out} be the node merged which does not belong to p_c.

$L_a(p_c) = l_b + comp(n_{out})$.

After the merger, CPL will increase by at least $comp(n_{out})$ and P_{crit} *might change.*

There are 2 possibilities:

1. If none of the execution paths p (p_c excluded) go through only one of the 2 nodes merged:
   $L_a(p) \leq L_b(p)$.
   Since $L_b(p) \leq l_b$, then $L_a(p) \leq l_b$.
   Hence CPL will increase by $comp(n_{out})$ and P_{crit} will not change after the merger.
2. If at least one execution path (p_c excluded) goes through only one of the 2 nodes merged:
   Let $p_1, p_2, \ldots, p_k$ be all the execution paths that go through only one of the 2 nodes merged (p_c excluded), $k \geq 1$.
   Let $n_{i,1}$ be the node merged which belongs to p_i, and $n_{i,2}$ be the node merged which does not belong to p_i.
   $L_a(p_i) = L_b(p_i) + comp(n_{i,2})$, $1 \leq i \leq k$.
   If $comp(n_{i,2}) \leq comp(n_{out})$ then P_{crit} will not change and CPL will increase by $comp(n_{out})$.
   If $n_{i,2} = n_{out}$ then P_{crit} will not change and CPL will increase by $comp(n_{out})$.
   Let p_m be the p_i such that $L_a(p_m)$ is the largest among all p_i's.

[8]Any edge in the graph belongs to at least one execution path.

There are 2 possible cases:

(a) If $L_a(p_i) \leq L_a(p_c)$, $1 \leq i \leq k$:
P_{crit} will not change and CPL will increase by $comp(n_{out})$.

(b) If at least one p_i is such that $L_a(p_i) > L_a(p_c)$:
After the merger,
$P_{crit} = p_m$ and $CPL = L_a(p_m)$.
CPL will increase by an amount greater than $comp(n_{out})$.
The increase in CPL is $L_b(p_m) + comp(n_{m,2}) - l_b$.

Case 2 P_{crit} **does not go through any of the 2 nodes merged:**
$L_a(p_c) = L_b(p_c) = l_b$.
CPL will either increase or remain unchanged.
P_{crit} *might change.*
There are 2 possibilities:

1. If no execution path p (p_c excluded) goes through only one of the 2 nodes merged:
$L_a(p) \leq L_b(p)$.
Since $L_b(p) \leq l_b$ then $L_a(p) \leq l_b$.
Thus, P_{crit} and CPL will not change.

2. If at least one execution path goes through only one of the 2 nodes merged:
Let $p_1, p_2, \ldots, p_k$ be all the execution paths that go through only one of the 2 nodes merged (p_c excluded), $k \geq 1$.
Let $n_{i,1}$ be the node merged which belongs to p_i, and $n_{i,2}$ be the node merged which does not belong to p_i.
$L_a(p_i) = L_b(p_i) + comp(n_{i,2})$, $1 \leq i \leq k$.
Hence, P_{crit} could change and CPL could increase.
There are 2 cases:

(a) If $L_a(p_i) \leq l_b$, $1 \leq i \leq k$:
P_{crit} and CPL will not change.

(b) If at least one p_i is such that $L_a(p_i) > l_b$:
Let p_m be the p_i such that $L_a(p_m)$ is the largest among all p_i's.
After the merger,
$P_{crit} = p_m$ and $CPL = L_a(p_m)$.
CPL will increase by $L_b(p_m) + comp(n_{m,2}) - l_b$.

Conclusion

- If $e \notin P_{crit}$ then CPL never decreases (it will either increase or remain unchanged) after the merger.
- If P_{crit} goes through only one of the 2 nodes merged, then CPL will increase by at least $comp(n_{out})$ after the merger, where n_{out} is the node merged which does not belong to P_{crit}.
- If P_{crit} does not go through any of the 2 nodes merged, then CPL will either increase or remain unchanged after the merger.

4.3 Merging Tasks: Effect of Parallelism Loss on CPL

In this section, we study the effect of parallelism loss on the CPL of the task graph. We consider two nodes n_1 and n_2 belonging to the task graph and connected by an edge $e = (n_1, n_2)$. We study the effect of merging nodes n_1 and n_2 on the CPL when the merger causes parallelism loss and when it doesn't.

4.3.1 No Parallelism Loss

Theorem: Assume that $ParSet(n_1) = ParSet(n_2)$, so that there is no parallelism loss when we merge n_1 and n_2.
Then the CPL of the task graph never increases as a result of the merger.

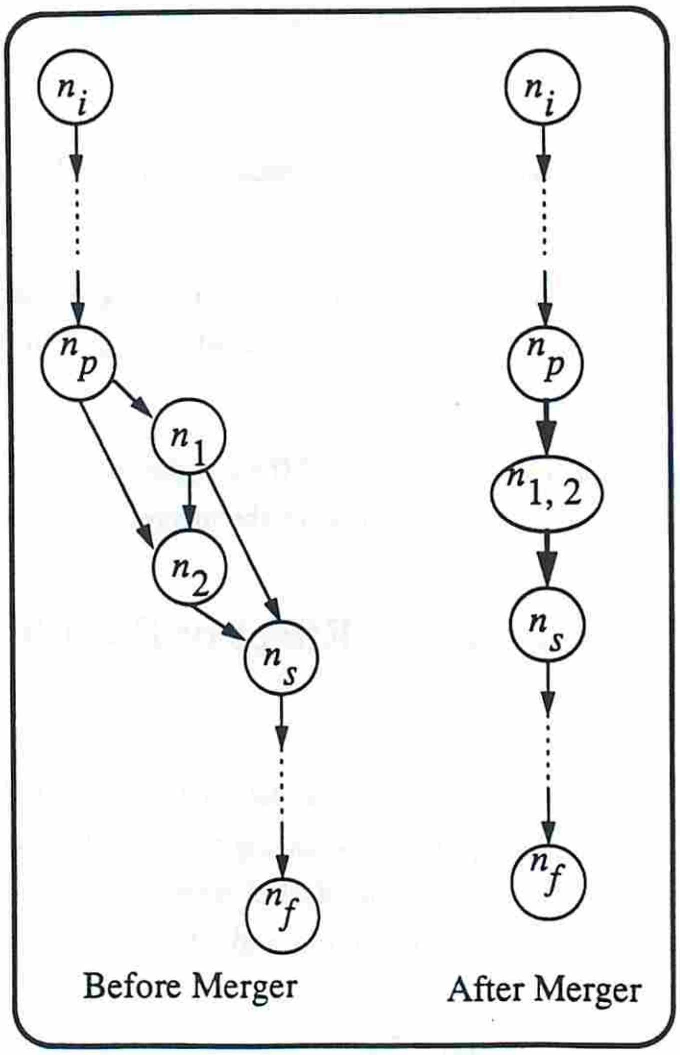


Figure 4.6: No parallelism loss

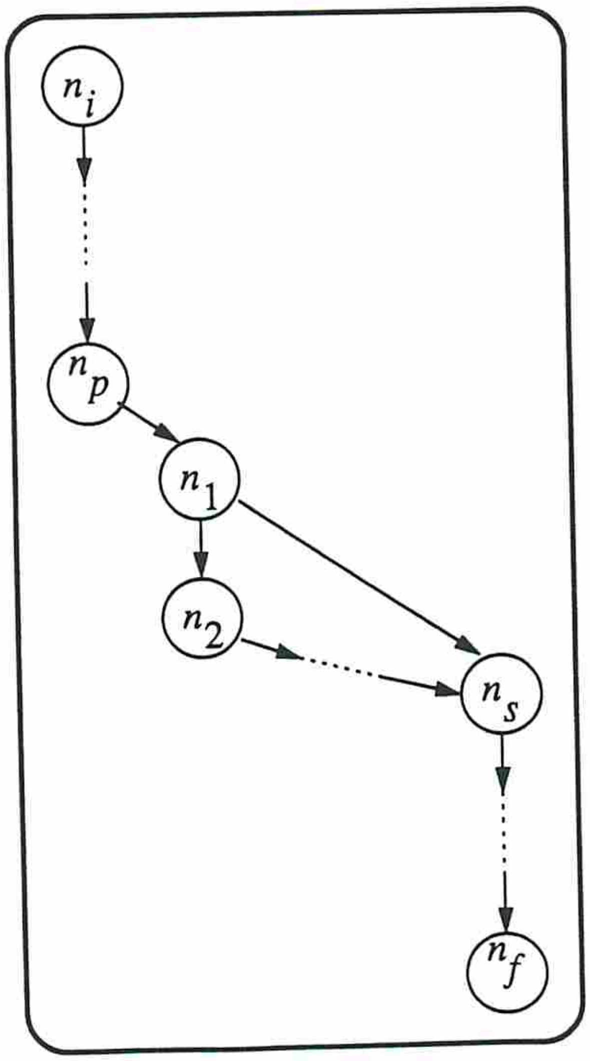


Figure 4.7: No parallelism loss

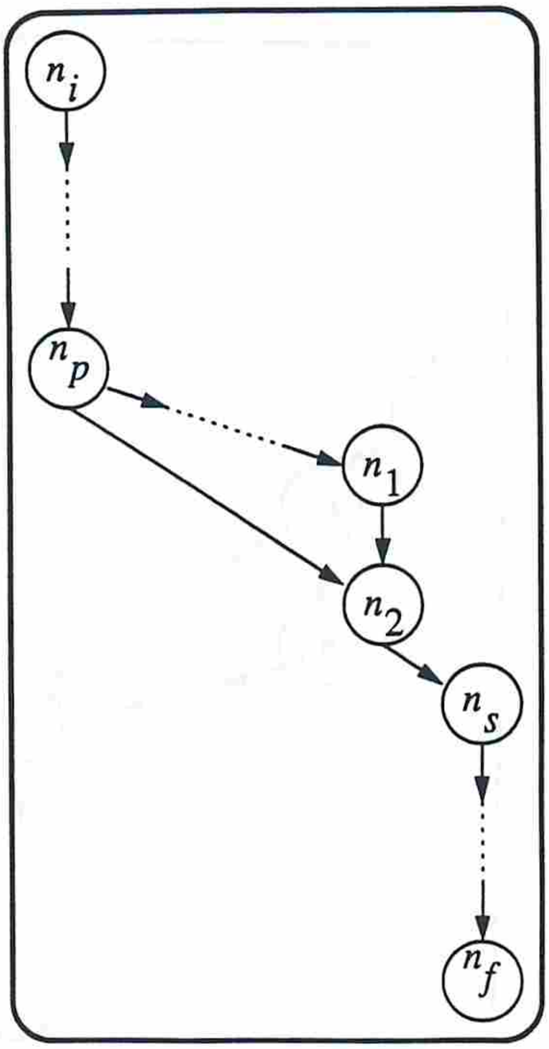


Figure 4.8: No parallelism loss

Proof

Let's use proof by contradiction. We assume that the CPL increases as a result of the merger. Therefore, there should exist at least one execution path p such that $L_a(p) > CPL_b$. Clearly, $L_b(p) \leq CPL_b$.

There are 3 possible cases:

1. p goes through both nodes n_1 and n_2.

2. p goes through n_1 but not n_2.

3. p goes through n_2 but not n_1.

The situation where p doesn't go through any of the 2 nodes n_1 and n_2 is not possible, since in that case $L(p)$ is not affected by the merger.

Let's investigate the 3 possible cases.

Case1 p goes through both nodes n_1 and n_2:
Let $p = (n_i, \ldots, n_p, n_1, n_2, n_s, \ldots, n_f)$.
For p to have the maximum increase in length after the merger, we have to have an edge (n_p, n_2) and an edge (n_1, n_s) (see figure 4.6).
Let $\Delta L := L_a(p) - L_b(p)$.
$\Delta L = delay(data(n_p, n_2)) + delay(data(n_1, n_s)) - comm(n_1, n_2)$.
The *comm* function includes both the start-up component and the delay component. Since in general the start-up component is much larger than the delay component, ΔL must be negative. Furthermore, in practice the edges (n_p, n_2) and (n_1, n_s) are most likely not to exist.
This means that $L_a(p) < L_b(p)$. Since $L_b(p) \leq CPL_b$, then $L_a(p) < CPL_b$.
This is a contradiction since we assumed that $L_a(p) > CPL_b$.

Case 2 p goes through n_1 and not n_2:
Let $p = (n_i, \ldots, n_p, n_1, n_s, \ldots, n_f)$.
Since $ParSet(n_1) = ParSet(n_2)$ and nodes n_1 and n_s are dependent, then nodes n_2 and n_s have to be dependent as well. Thus either there exists a path from n_2 to n_s or there exists a path from n_s to n_2. If there exists a path from n_s to n_2 then there exists a path from n_1 to n_2 other than (n_1, n_2).

Therefore we will have a cycle in the task graph after the merger. Hence we have to disregard this case, which means that there exists a path from n_2 to n_s (see figure 4.7).

Let $p' = (n_i, \ldots, n_p, n_1, n_2, \ldots, n_s, \ldots, n_f)$.

Let $\Delta L := L_a(p) - L_b(p')$.

For ΔL to have its maximum value, $L_a(p)$ has to be maximized and $L_b(p)$ has to be minimized. Hence, the path from n_2 to n_s has to be constituted of the single edge (n_2, n_s).

Also for $L_a(p)$ to be maximized, we have to have an edge (n_p, n_2) (see figure 4.6).

Therefore, $p' = (n_i, \ldots, n_p, n_1, n_2, n_s, \ldots, n_f)$.

Hence,

$\Delta L = delay(data(n_p, n_2)) + delay(data(n_1, n_s)) - comm(n_1, n_2)$.

Again, ΔL must be negative. Furthermore, in practice the edges (n_p, n_2) and (n_2, n_s) are most likely not to exist.

This means that $L_a(p) < L_b(p')$. Since $L_b(p') \leq CPL_b$, then $L_a(p) < CPL_b$. This is a contradiction since we assumed that $L_a(p) > CPL_b$.

Case 3 p goes through n_2 and not n_1:

Let $p = (n_i, \ldots, n_p, n_2, n_s, \ldots, n_f)$.

Since $ParSet(n_1) = ParSet(n_2)$ and nodes n_2 and n_p are dependent, then nodes n_1 and n_p have to be dependent as well. Thus either there exists a path from n_1 to n_p or there exists a path from n_p to n_1. If there exists a path from n_1 to n_p then there exists a path from n_1 to n_2 other than (n_1, n_2). Therefore we will have a cycle in the task graph after the merger. Hence we have to disregard this case, which means that there exists a path from n_p to n_1 (see figure 4.8).

Let $p' = (n_i, \ldots, n_p, \ldots, n_1, n_2, n_s, \ldots, n_f)$.

Let $\Delta L := L_a(p) - L_b(p')$.

For ΔL to have its maximum value, $L_a(p)$ has to be maximized and $L_b(p)$ has to be minimized. Hence, the path from n_p to n_1 has to be constituted of the single edge (n_p, n_1).

Also for $L_a(p)$ to be maximized, we have to have an edge (n_1, n_s) (see figure

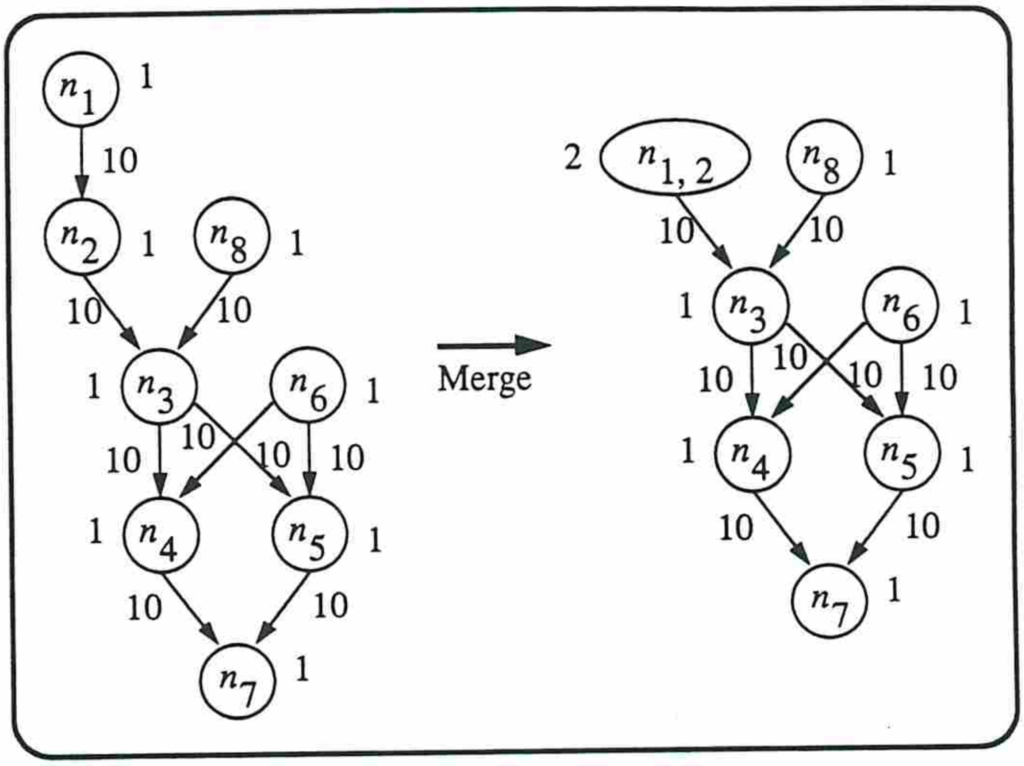


Figure 4.9: Example: no parallelism loss

4.6).

Therefore, $p' = (n_i, \ldots, n_p, n_1, n_2, n_s, \ldots, n_f)$.

Hence,

$\Delta L = delay(data(n_p, n_2)) + delay(data(n_1, n_s)) - comm(n_1, n_2)$.

Again, ΔL must be negative. Furthermore, in practice the edges (n_p, n_1) and (n_1, n_s) are most likely not to exist.

This means that $L_a(p) < L_b(p')$. Since $L_b(p') \leq CPL_b$, then $L_a(p) < CPL_b$.

This is a contradiction since we assumed that $L_a(p) > CPL_b$.

Hence, there cannot exist an execution path p such that $L_a(p) > CPL_b$

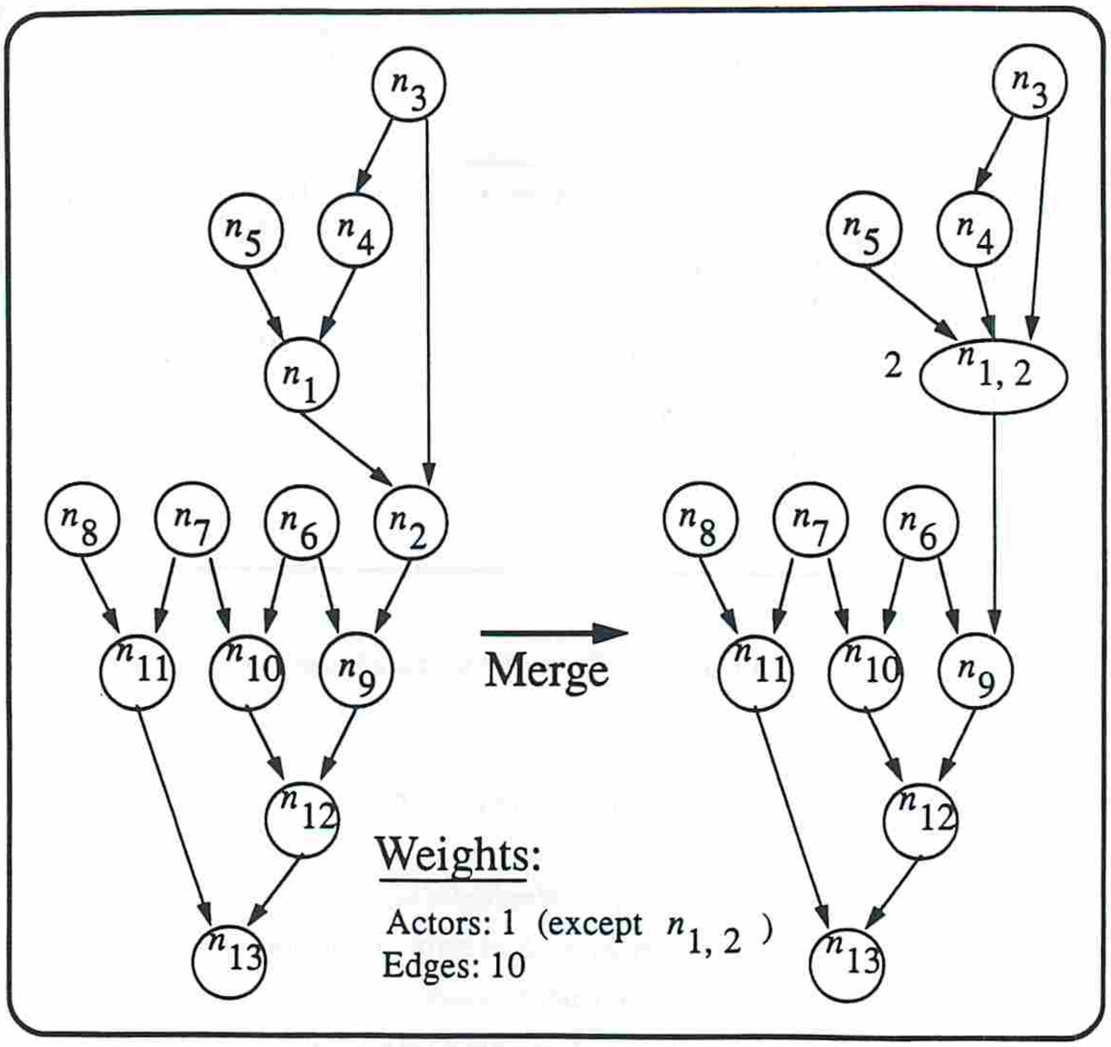


Figure 4.10: Example: no parallelism loss

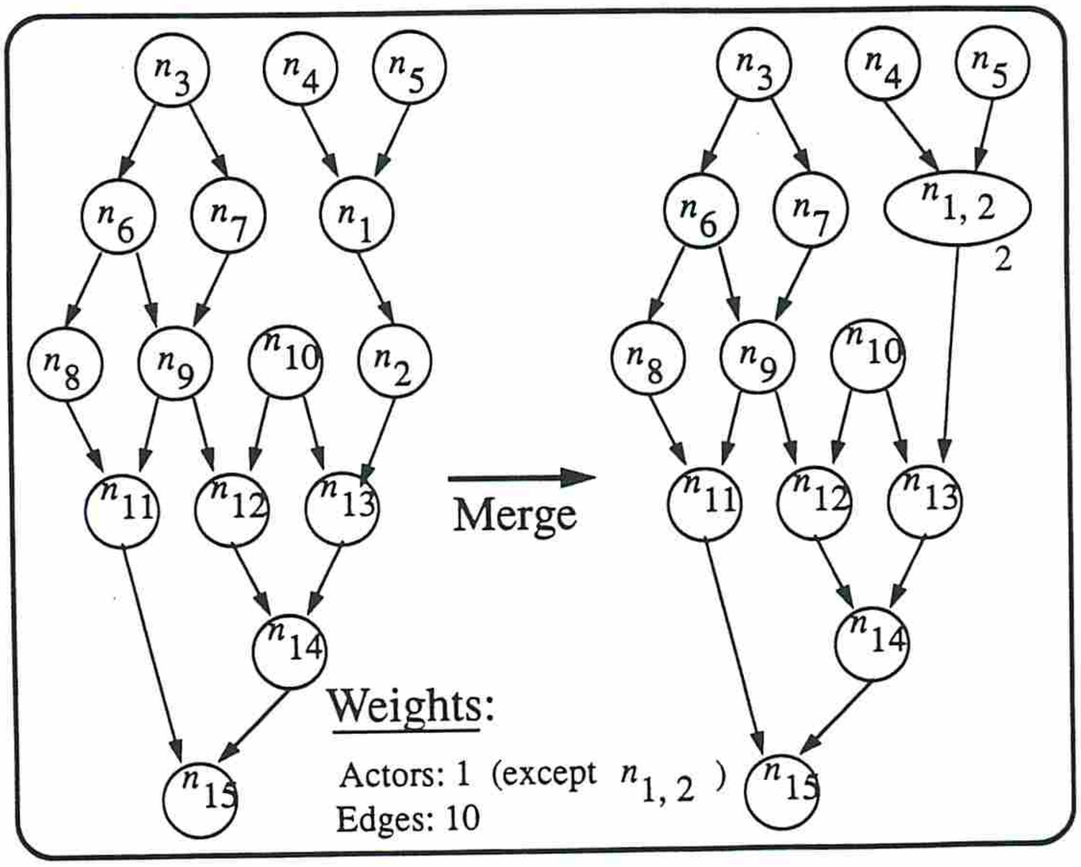


Figure 4.11: Example: no parallelism loss

Examples

- Figure 4.9 shows a task graph before and after the merger.
  $ParSet(n_1) = \{n_6, n_8\}$
  $ParSet(n_2) = \{n_6, n_8\}$
  $ParSet(n_1) = ParSet(n_2)$
  Hence, there is no parallelism loss as a result of the merger.
  Before merger: CPL = 45.
  After merger: CPL = 35.
  The CPL has decreased.

- Figure 4.10 shows a task graph before and after the merger.
  $ParSet(n_1) = \{n_6, n_7, n_8, n_{10}, n_{11}\}$
  $ParSet(n_2) = \{n_6, n_7, n_8, n_{10}, n_{11}\}$
  $ParSet(n_1) = ParSet(n_2)$
  Hence, there is no parallelism loss as a result of the merger.
  Before merger: CPL = 67.
  After merger: CPL = 57.
  The CPL has decreased.

- Figure 4.11 shows a task graph before and after the merger.
  $ParSet(n_1) = \{n_3, n_6, n_7, n_8, , n_9, n_{10}, n_{11}, n_{12}\}$
  $ParSet(n_2) = \{n_3, n_6, n_7, n_8, , n_9, n_{10}, n_{11}, n_{12}\}$
  $ParSet(n_1) = ParSet(n_2)$
  Hence, there is no parallelism loss as a result of the merger.
  Before merger: CPL = 56.
  After merger: CPL = 56.
  The CPL has not changed.

4.3.2 There is Parallelism Loss

Theorem: Assume that there is parallelism loss when we merge n_1 and n_2. Then the CPL of the task graph could increase as a result of the merger.

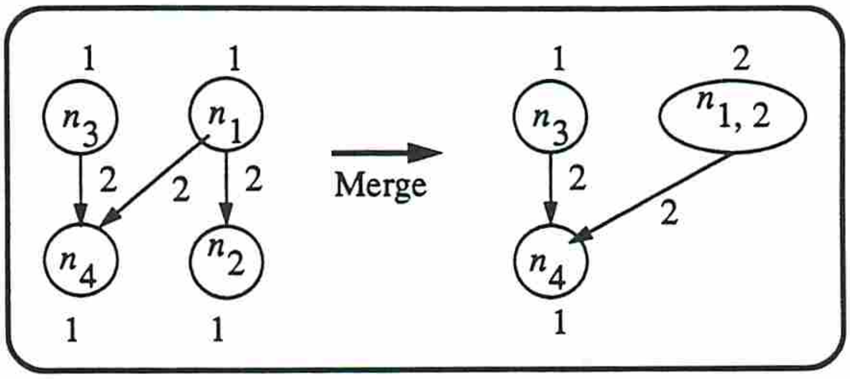


Figure 4.12: Example: there is parallelism loss

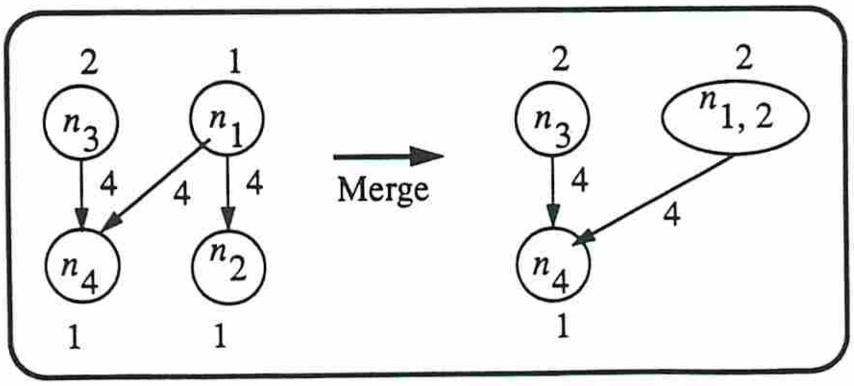


Figure 4.13: Example: there is parallelism loss

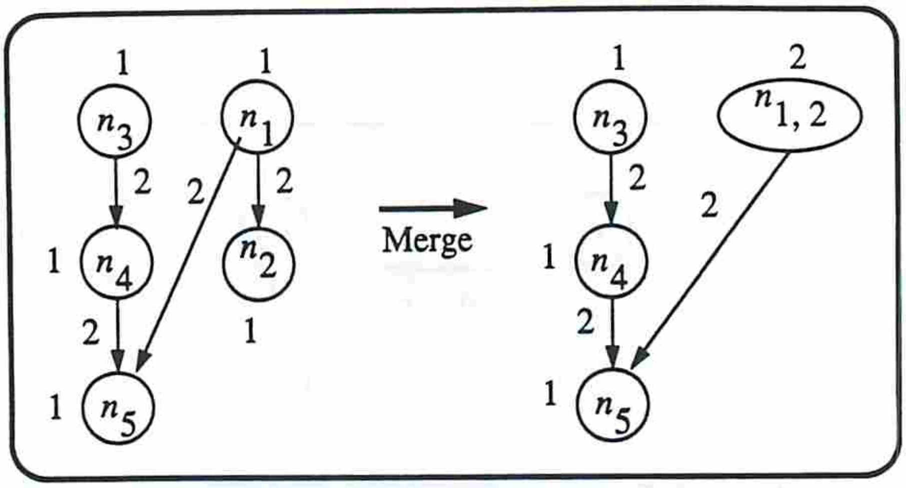


Figure 4.14: Example: there is parallelism loss

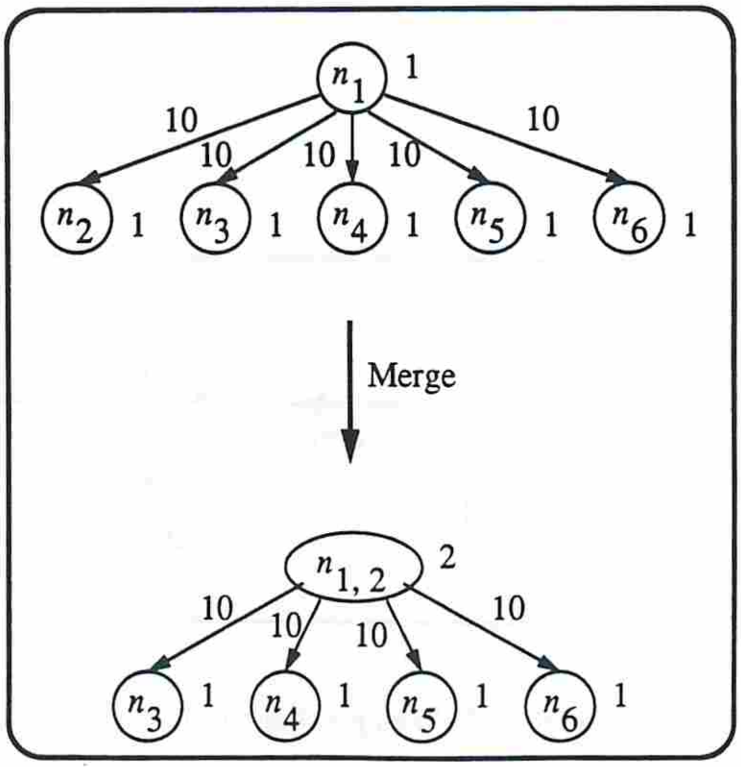


Figure 4.15: Example: there is parallelism loss

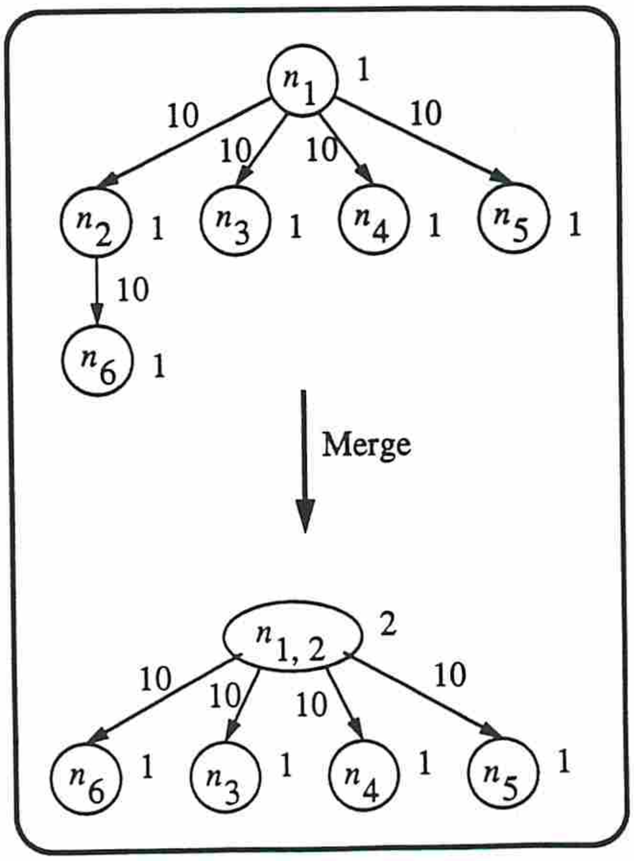


Figure 4.16: Example: there is parallelism loss

Proof

Let's prove the claim in the theorem by studying some examples.
In all the figures used in the following examples, the number next to a node represents its execution time, and the number next to an edge represents the communication time caused by the edge.

- Figure 4.12 shows a task graph before and after the merger.
  $ParSet(n_2) = \{n_3, n_4\}$
  $DepSet(n_1) = \{n_2, n_4\}$
  $ParSet(n_2) \cap DepSet(n_1) = \{n_4\}$
  Hence, there is parallelism loss with respect to n_2.
  Before merger: CPL = PARTIME = 4.
  After merger: CPL = 5.
  Thus the CPL has increased.
  Note that before the merger, the graph had a critical path which contained n_1 and not n_2 ((n_1, n_4)), and that is why we have an increase in the CPL.

- Figure 4.13 shows the same graph as in figure 4.12, except for the weights.
  Before merger: CPL = 7.
  After merger: CPL = 7.
  Thus the CPL did not change.

- Figure 4.14 shows a task before and after the merger.
  $ParSet(n_2) = \{n_3, n_4, n_5\}$
  $DepSet(n_1) = \{n_2, n_5\}$
  $ParSet(n_2) \cap DepSet(n_1) = \{n_5\}$
  Hence, there is parallelism loss with respect to n_2.
  Before merger: CPL = 7.
  After merger: CPL = 7.
  Thus the CPL did not change.

- Figure 4.15 shows a task graph before and after the merger.
  $ParSet(n_2) = \{n_3, n_4, n_5, n_6\}$: independent set.
  $DepSet(n_1) = \{n_2, n_3, n_4, n_5, n_6\}$

$ParSet(n_2) \cap DepSet(n_1) = \{n_3, n_4, n_5, n_6\} = ParSet(n_2)$
Hence, there is parallelism and usable parallelism loss with respect to n_2.
Before merger: CPL = 12.
After merger: CPL = 13.
Thus the CPL has increased.
Note that before the merger, the graph had at least one critical path which contained n_1 and not n_2 (e.g. (n_1, n_3)), and that is why we have an increase in the CPL.

- Figure 4.16 shows a task graph before and after the merger.
  $ParSet(n_2) = \{n_3, n_4, n_5\}$: independent set.
  $DepSet(n_1) = \{n_2, n_3, n_4, n_5, n_6\}$
  $ParSet(n_2) \cap DepSet(n_1) = \{n_3, n_4, n_5\} = ParSet(n_2)$
  Hence, there is parallelism and usable parallelism loss with respect to n_2.
  Before merger: CPL = 23.
  After merger: CPL = 13.
  The CPL has decreased.

Conclusion

- Given a task graph, if there is parallelism loss as a result of task merging, the CPL can increase, remain unchanged, or decrease.
- Given a task graph, if there is usable parallelism loss as a result of task merging, the CPL can increase, remain unchanged, or decrease.

4.4 A Comparison with DSC

The scheduling problem as defined by Tao Yang [18, 58, 59] (a sequence of task clustering) has some similarities with the way we define the partitioning problem (a sequence of task merging). Mainly both assume the availability of an infinite number of PEs and non-zero communication overhead between PEs. As a consequence, both problems use the CPL of the task graph as the parallel execution time.

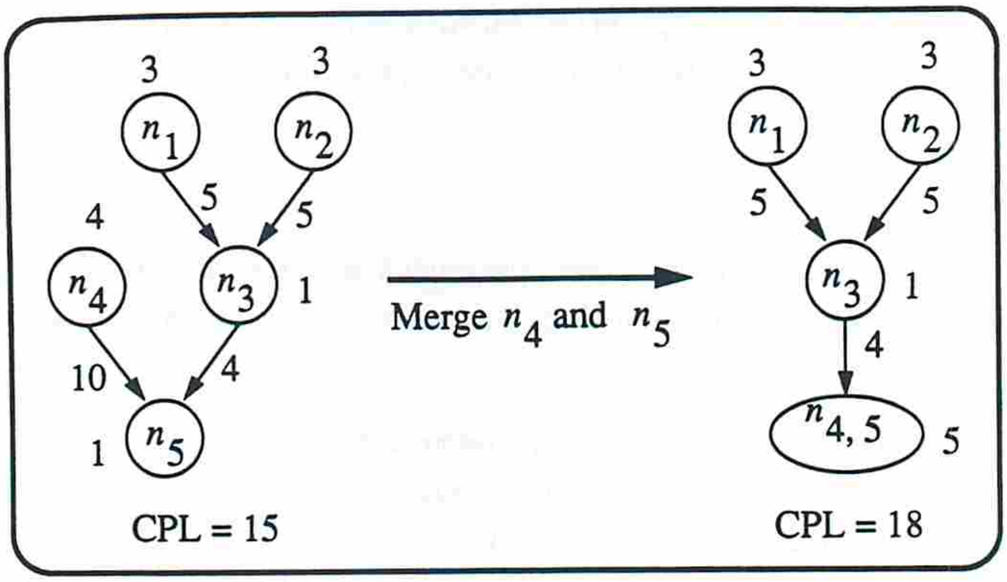


Figure 4.17: Example of task merging

However, there is a major difference, since merging tasks involves changes in the task graph[9] whereas task clustering doesn't[10].

Because of this, there is a major difference in the effect of task merging and task clustering on the length of execution paths and as a consequence on the CPL of the task graph.

4.4.1 Task Merging

As we saw previously, task merging could increase the CPL of the task graph. As an example, consider the task graph in figure 4.17. Before the merger, the critical path is (n_4, n_5) and the CPL is 15. After the merger, both execution paths (n_1, n_3, n_5) and (n_2, n_3, n_5) increase in length by 4, since they both go through

[9]When two tasks are merged, they are replaced by a new task and some edges are replaced by new ones.

[10]Clustering simply means that all tasks in the same cluster are executed in the same PE. The only change in the task graph is the addition of pseudo-edges between independent tasks in the same cluster to impose an execution order in the PE. Also all weights of edges between tasks in the same cluster are zeroed.

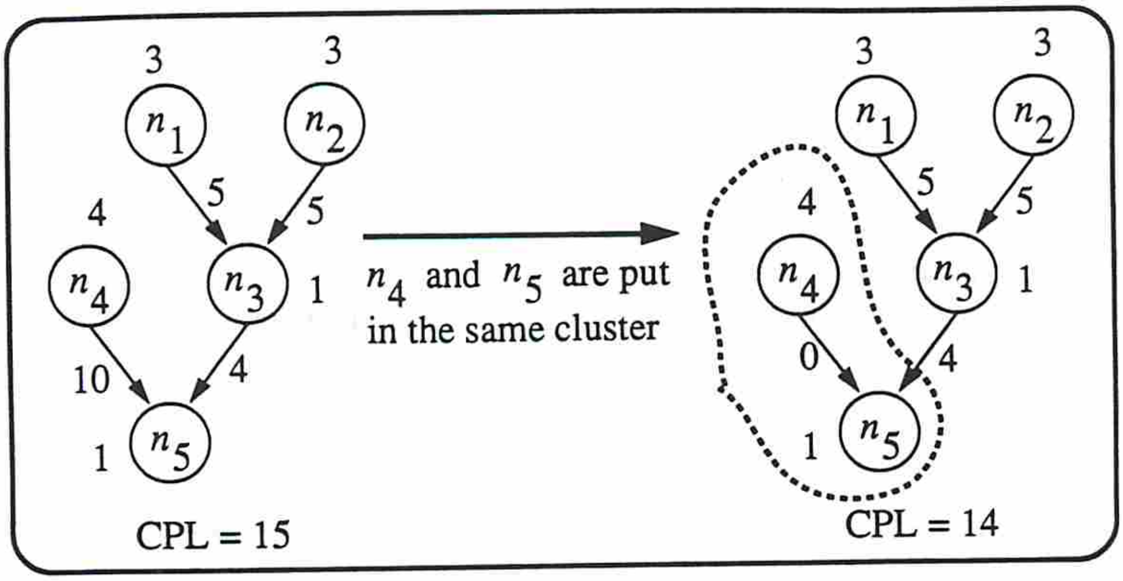


Figure 4.18: Example of task clustering

only one of the 2 nodes merged. The CPL increases to 18. This is a side effect of the merger.

4.4.2 Task Clustering

Task clustering rarely causes the CPL of the task graph to increase. As an example, consider the task graph in figure 4.18. Before the merger, the critical path is (n_4, n_5) and the CPL is 15. After n_4 and n_5 are put in the same cluster, the execution paths (n_1, n_3, n_5) and (n_2, n_3, n_5) are not affected[11]. The CPL decreases to 14.

4.4.3 Consequence

Because of the side effect caused by task merging, reducing the CPL using task merging is much harder than using task clustering. This makes the partitioning

[11]The only case when execution paths could increase in length is when pseudo-edges are added.

problem (as defined in this work) much harder than the scheduling problem (as defined by Tao Yang).

4.5 Criteria for Merging

In this section, we list some criteria that will be used by the partitioning heuristics to choose the edge to be merged.
We use the results of the previous analysis from the previous sections to obtain these criteria.

- Edge has to belong to a critical path.
- Edge that belongs to all execution paths (if such an edge exists).
- Edge e such that none of the execution paths go through only one of the 2 nodes connected by e.
- Edge $e = (n_1, n_2)$ such that $ParSet(n_1) = ParSet(n_2)$ (no parallelism loss as a result of the merger).
- Edge e with the largest $comm(e)$. This way all execution paths which go through e will decrease in length by a maximum quantity.
- Edge $e = (n_1, n_2)$ such that $comp(e)$ is smallest. This way, if there is an execution path p that goes through only one of the two nodes n_1 and n_2, the length of p increases by the smallest possible quantity.
- Edge $e = (n_1, n_2)$ such that the merger causes the minimum loss in parallelism:
  $(|ParSet(n_1)| - |ParSet(n_{1,2})|) + (|ParSet(n_2)| - |ParSet(n_{1,2})|)$
  is the smallest.
- Edge $e = (n_1, n_2)$ such that the merger causes the minimum loss in usable parallelism:
  $(MaxPar(n_1) - MaxPar(n_{1,2})) + (MaxPar(n_2) - MaxPar(n_{1,2}))$
  is the smallest.

- Edge e such that the merger causes the least amount of execution paths to increase in length.
  For instance, we could choose edge $e = (n_1, n_2)$ such that the number of execution paths that go through only one of the 2 nodes n_1 and n_2 is minimum.

- Edge e such that the merger causes the largest number of execution paths to decrease in length.
  In other words, we look for edge e such that the number of execution paths that go through e is maximum.

We have to make sure that the criteria used in our partitioning algorithm are not too costly. For instance, the 2 last criteria mentioned above require a large time complexity.

Chapter 5

The Partitioning Heuristics

Note: In the partitioning algorithm, if there is more than one critical path, then choose one randomly.

Safe Edges: Let g be a task graph (DAG). An edge $e = (n_1, n_2)$ is said to be a *safe edge* if merging nodes n_1 and n_2 does not cause any cycles to be created in g. Otherwise, e is said to be an *unsafe edge.*

A Requirement: An edge $e = (n_1, n_2)$ in a task graph is chosen for merger *if and only if* e is safe. In other words, there should not exist a path from n_1 to n_2 other than (n_1, n_2).

Lemma: Let g be a task graph (DAG) and $e = (n_1, n_2)$ be a safe edge. If a path p in g goes through both nodes n_1 and n_2, then $e \in p$.

Proof

Assume a path p in g goes through both nodes n_1 and n_2.
If $e \notin p$ then there are 2 possibilities:

1. There is a path from n_1 to n_2 other than (n_1, n_2). This contradicts our assumption that e is a safe edge.

2. There is a path from n_2 to n_1. This means that g has a cycle, which contradicts our assumption that g is a DAG.

Hence, $e \in p$.

Perfect Edges: We define a *perfect edge* to be one that belongs to all execution paths in the DAG. Otherwise, the edge is said to be an *imperfect edge.*

Risky Edges: An edge $e = (n_1, n_2)$ is said to be a *risky edge* if there exists at least one execution path that goes through only one of the nodes n_1 and n_2.

5.0.1 Heuristics

In what follows, we show a few heuristics that can be used to choose the edge to be merged during each iteration of the partitioning algorithm. Since these heuristics are used in each merging step (i.e. merging iteration), we also call them merging heuristics.

Heuristic 1

1. Find heaviest safe edge e in task graph which is *perfect.*
   If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.
   If no such edge go to 2, else go to 5.

2. Find heaviest safe edge e in P_{crit} which is not *risky*. If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.
   If no such edge go to 3, else go to 5.

3. Find Heaviest safe edge $e = (n_1, n_2) \in P_{crit}$ such that
   $\text{ParSet}(n_1) = \text{ParSet}(n_2)$.
   If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.
   If no such edge go to 4, else go to 5.

4. Find safe edge $e = (n_1, n_2) \in P_{crit}$ such that
   $(|ParSet(n_1)| - |ParSet(n_{1,2})|) + (|ParSet(n_2)| - |ParSet(n_{1,2})|)$

is the smallest among all safe edges in P_{crit}.
If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.

5. Merge 2 tasks linked by e.

Heuristic 2

1. Find Heaviest safe edge $e = (n_1, n_2) \in P_{crit}$ such that
   ParSet(n_1) = ParSet(n_2).
   If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.
   If no such edge go to 2, else go to 3.

2. Find safe edge $e = (n_1, n_2) \in P_{crit}$ such that
   $(|ParSet(n_1)| - |ParSet(n_{1,2})|) + (|ParSet(n_2)| - |ParSet(n_{1,2})|)$
   is the smallest among all safe edges in P_{crit}.
   If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.

3. Merge n_1 and n_2.

Heuristic 3

1. Find Heaviest safe edge $e = (n_1, n_2) \in P_{crit}$ such that
   ParSet(n_1) = ParSet(n_2).
   If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.
   If no such edge go to 2, else go to 3.

2. Find safe edge $e = (n_1, n_2) \in P_{crit}$ such that
   $(MaxPar(n_1) - MaxPar(n_{1,2})) + (MaxPar(n_2) - MaxPar(n_{1,2}))$
   is the smallest among all safe edges in P_{crit}.
   If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.

3. Merge n_1 and n_2.

Finding $MaxPar(n_{1,2})$ without doing the merger

ParSet($n_{1,2}$) = ParSet(n_1) $\cap$ ParSet(n_2).

Or better yet: Express $MaxPar(n_{1,2})$ in terms of $MaxPar(n_1)$ and $MaxPar(n_2)$.

Heuristic 4

1. Find Heaviest safe edge $e = (n_1, n_2) \in P_{crit}$ such that
   ParSet(n_1) = ParSet(n_2).
   If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.
   If no such edge go to 2, else go to 4.

2. Find Heaviest safe edge $e = (n_1, n_2)$ in task graph such that
   ParSet(n_1) = ParSet(n_2).
   If there is more than one such edge e, choose the one such that $comp(e)$ has the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.
   If no such edge go to 3, else go to 4.

3. Find safe edge $e = (n_1, n_2)$ in graph, such that
   $(|ParSet(n_1)| - |ParSet(n_{1,2})|) + (|ParSet(n_2)| - |ParSet(n_{1,2})|)$
   is the smallest among all safe edges.
   If there is more than one such edge e, choose the one such that $comp(e)$ has

the minimal value. If there is still more than one edge that satisfies that, then choose one randomly.

4. Merge n_1 and n_2.

Heuristic 5

1. Find safe edge e in P_{crit} that has the largest *merge merit.*
   If there is more than one such edge e, then choose one randomly.

2. Merge 2 tasks linked by e.

Merge Merit of an Edge: The merge merit of an edge e is
$merge(e) := \alpha.comm(e) - \beta.comp(e)$, $\alpha, \beta > 0$.

Determining α and β

$\frac{\alpha}{\beta} \Re R^{comm}_{comp}$

R^{comm}_{comp} := communication to computation ratio of target machine.

Remarks

- Heuristic 4, Step2: if $(n_1, n_2) \notin P_{crit}$ then no decrease in CPL !!

- Heuristics 3 and 4 have very high time complexities.

- Heuristics 1 and 2 have the lowest time complexities. Heuristic 2 is less costly than heuristic 1, but heuristic 1 is more efficient than heuristic 2.

- We chose heuristic 1 to do the performance analysis of our partitioning algorithm.

5.1 Some Properties

The following properties enable us to reduce the time complexities of the partitioning heuristics, by making it easier and quicker to find the edge to be merged.

Theorem: For any DAG g such that each node in the graph has at most one output, all edges in g are *safe.*

Proof

We use proof by contradiction.
Assume that there exists an edge $e = (n_1, n_2) \in g$ such that e is unsafe.
$\Leftrightarrow$ There exists at least one path p from n_1 to n_2 other than (n_1, n_2).
Let $p = (n_1, n_{x1}, n_{x2}, \ldots, n_{xm}, n_2)$, $m \geq 1$.
There are 2 possibilities:

1. $n_{x1} = n_2$. Therefore g is cyclic: contradiction.

2. $n_{x1} \neq n_2$. Therefore n_1 has at least 2 outputs: contradiction.

Therefore there cannot be any unsafe edges in g.

Lemma: Given a task graph (DAG) and a safe edge $e = (n_1, n_2)$ in the graph. e is not a *risky* edge
$\Longleftrightarrow$
Node n_1 has *only one* output edge (e), and node n_2 has *only one* input edge (e).

Proof

1. Assume that No execution path goes through *only one* of the 2 nodes connected by edge e.

   - If n_1 has more than one output edge (let the other output edge be $e' = (n_1, n_3)$), then there is at least one execution path p that goes through edge (n_1, n_3). Clearly, p cannot go through edge (n_1, n_2) (otherwise p will have a cycle, which means that the graph is not acyclic). Thus, p cannot go through n_2 (otherwise p goes through both nodes n_1 and n_2, which implies that it goes through edge e). Hence p goes through n_1 and not n_2. This contradicts our assumption. Therefore, n_1 has only one output edge.

- If n_2 has more than one input edge (let the other input edge be $e' = (n_3, n_2)$), then there is at least one execution path p that goes through edge (n_3, n_2). Clearly, p cannot go through edge (n_1, n_2) (otherwise p will have a cycle, which means that the graph is not acyclic). Thus, p cannot go through n_1 (otherwise p goes through both nodes n_1 and n_2, which implies that it goes through edge e). Hence p goes through n_2 and not n_1. This contradicts our assumption. Therefore, n_2 has only one input edge.

2. Assume that node n_1 has *only one* output edge (e), and node n_2 has *only one* input edge (e).

   - Any execution path p that goes through n_1 has to go through edge e (since n_1 has only one output edge).
   - Any execution path p that goes through n_2 has to go through edge e (since n_2 has only one input edge).

   Hence, no execution path goes through only n_1 or only n_2.

Corollary: Given a task graph (DAG) such that each node in the graph has *at most one output*, and a safe edge $e = (n_1, n_2)$ in the graph.
e is not a *risky* edge
$\Longleftrightarrow$
Node n_2 has *only one* input edge (e).

Proof: Trivial, from the previous lemma.

Lemma: Given a task graph (DAG) and an edge $e = (n_1, n_2)$ in the graph.
e is a *perfect* edge
$\Longrightarrow$

Node n_1 has *only one* output edge (e), and node n_2 has *only one* input edge (e)
AND
$ParSet(n_1) = ParSet(n_2) = \emptyset$

Proof

Assume that e is a perfect edge.
If n_1 has more than one output edge or n_2 has more than one input edge, then clearly there exists at least one execution path that doesn't go through e. Hence, e is not a perfect edge, which contradicts our assumption. Therefore, node n_1 has only one output edge, and node n_2 has only one input edge.
Now, let's prove that $ParSet(n_1) = ParSet(n_2) = \emptyset$.
We know that all execution paths go through e. For any node n in the graph other than n_1 and n_2, n belongs to at least one execution path p. Since p goes through e, then p goes through n_1 and n_2. Hence, n and n_1 are dependent and n and n_2 are dependent. Therefore $ParSet(n_1) = ParSet(n_2) = \emptyset$.

Lemma: Given a task graph (DAG) such that each node in the graph has *at most one output*, and an edge $e = (n_1, n_2)$ in the graph.
$\text{ParSet}(n_1) = \text{ParSet}(n_2) \iff$
Node n_2 has *only one* input edge (e).

Proof

1. Assume that $\text{ParSet}(n_1) = \text{ParSet}(n_2)$.
   If n_2 has more than one input edge, then it will have at least one input edge (n_3, n_2) other than e.
   There can be no path between n_1 and n_3, otherwise we must have a path from n_2 to n_3 (since n_1 has only one output edge, which is (n_1, n_2)), which means that the task graph is acyclic.
   There can be no path between n_3 and n_1, otherwise we must have a path from n_2 to n_1 (since n_3 has only one output edge, which is (n_3, n_2)), which means that the task graph is acyclic.
   Hence $n_3 \in ParSet(n_1)$. Clearly, $n_3 \notin ParSet(n_2)$. Therefore, $ParSet(n_1) \neq ParSet(n_2)$, which contradicts our original assumption.
   Therefore, n_2 has only one input edge.

2. Assume that n_2 has only one input edge.

- $\forall n \in ParSet(n_1)$, there is no path from n_1 to n and no path from n to n_1.
  There cannot be a path from n to n_2, otherwise we must have a path from n to n_1 (since n_2 has only one input edge).
  There cannot be a path from n_2 to n, otherwise we must have a path from n_1 to n (because of edge (n_1, n_2)).
  Hence, $n \in ParSet(n_2)$.
  Thus, $ParSet(n_1) \subset ParSet(n_2)$.
- $\forall n \in ParSet(n_2)$, there is no path from n_2 to n and no path from n to n_2.
  There cannot be a path from n_1 to n, otherwise we must have a path from n_2 to n (since n_1 has only one output edge).
  There cannot be a path from n to n_1, otherwise we must have a path from n to n_2 (because of edge (n_1, n_2)).
  Hence, $n \in ParSet(n_1)$.
  Thus, $ParSet(n_2) \subset ParSet(n_1)$.

Therefore, $ParSet(n_1) = ParSet(n_2)$.

Corollary: Given a task graph (DAG) such that each node in the graph has *at most one output*, and a safe edge $e = (n_1, n_2)$ in the graph.
$\mathrm{ParSet}(n_1) = \mathrm{ParSet}(n_2) \iff$
e is not a *risky* edge.

Proof: From a previous corollary and a previous lemma.

5.2 Time Complexity of Partitioning Algorithm

Let E be the number of edges and N be the number of nodes in the program graph.
2, the initial task graph will have N nodes and at most E edges.

5.2.1 DAG Traversal

As will be seen later, the partitioning algorithm requires traversal of the task graph, which is a DAG.
In what follows, we describe the general procedure for DAG traversal.

```
Let $Q$ be a queue (could be implemented as a linked list).
$Q \leftarrow \emptyset$.
Insert all root nodes in $Q$ (in any order).
Repeat until $Q = \emptyset$

    $n \leftarrow$ Front of $Q$.
    Delete $n$ from $Q$.
    visit($n$) % Node $n$ is visited here.
    IF $n$ is not a leaf node THEN

        Insert all children of $n$ in $Q$
        % The way insertion is done depends on the traver-
        sal
        % type (e.g. breadth-first, depth-first).
```

Traversal of general DAGs is different from tree traversal. With general DAGs, if we are not careful, a node might be visited more than once. Clearly, this is not the case for trees. The reason for this is that for general DAGs, a node may have more than one input edge.
In order to avoid visiting nodes more than once, when a node is put in the queue Q, it is marked as *queued*. After a node is visited, only its children which are not marked *queued* are inserted in Q.
Hence the correct version of the general algorithm is as follows.

```
Let $Q$ be a queue (could be implemented as a linked list).
$Q \leftarrow \emptyset$.
Insert all root nodes in $Q$ (in any order)
% No need to mark root nodes as queued.
Repeat until $Q = \emptyset$
```

```
        $n \leftarrow$ Front of $Q$.
        Delete $n$ from $Q$.
        visit($n$) % Node $n$ is visited here.
        $n$ is marked *visited*. % This marking may not be needed.
        IF $n$ is not a leaf node THEN
            Insert all children of $n$ that are not marked *queued*
            in $Q$, and mark them as *queued* % So that nodes
            are not visited more than once.
            % The way insertion is done depends on traversal
            % type (e.g. breadth-first, depth-first).
```

Deadlock Situations:
The algorithm for DAG traversal listed above never leads to deadlock situations (deadlock means that the algorithm ends and there are still nodes not visited). To see why this is the case, assume that during the execution of the algorithm we reach a deadlock situation. This means that the queue Q is empty and there is at least one node n in the graph that hasn't been visited yet. Clearly, n hasn't been inserted in Q yet. Therefore, none of its parent nodes has been visited yet. Let n_1 be a parent node of n (if any). Then n_1 was never inserted in Q either. This goes on until we reach a root node r (since the graph is acyclic), and establish that r was never inserted in Q. Clearly this cannot be the case since all root nodes are inserted in Q at the beginning of the algorithm. Hence our assumption that there is a deadlock situation cannot be true.
Another way to see why we cannot have any deadlock situations is to notice that starting from the root nodes, we can reach any node in the graph by following the paths emanating from the root nodes.

Depth-First Traversal

For depth-first traversal, the children of the node just visited are inserted at the Front of Q. The order among the children nodes in Q does not matter.

Breadth-First Traversal

For breadth-first traversal, the children of the node just visited are inserted at the Rear of Q. The order among the children nodes in Q does not matter.

Remarks

1. Breadth-first and depth-first traversals for general DAGs are different from the ones for trees. This is so because for general DAGs a node may have more than one input edge.

2. Time Complexity:
   Breadth-first and depth-first traversals take at the most $O(E + N)$ time complexity.
   Proof:
   All nodes in the graph are inserted in Q exactly once and are visited exactly once. This takes $O(N)$ time.
   Each time a node n is visited, each one of its children is examined to see whether it is marked *queued* or not. The number of children of n is equal to the number of output edges of n. Therefore, since each node in the graph is visited exactly once, this takes $O(E)$ time.

Parents-First Traversal

In parents-first traversal, a node is not visited until all of its parent nodes are visited[1]. The idea here is to keep a counter for each node in the graph (except the root nodes). This counter is used to keep track of the number of parent nodes of a given node that are already visited. When the counter of some node n is equal to the total number of parents of node n, then we know that all the parent nodes of n are already visited. A child node is inserted in the queue Q only when all of its parents are already visited.
The procedure is as follows:

[1]This is a traversal using a topological order.

```
Let $Q$ be a queue (could be implemented as a linked list).
$Q \leftarrow \emptyset$.
Insert all root nodes in $Q$ (in any order).
FOR all non-root nodes $n$ in the graph DO

    % Initialize the counters of the nodes.
    $n.count \leftarrow$ Number of parent nodes of $n$

Repeat until $Q = \emptyset$

    $n \leftarrow$ Front of $Q$.
    Delete $n$ from $Q$.
    visit($n$) % Node $n$ is visited here.
    $n$ is marked *visited.* % This marking may not be needed.
    IF $n$ is not a leaf node THEN
        FOR all children nodes $n'$ of $n$ DO
            $n'.count \leftarrow n'.count - 1$ % One more parent
            visited.
            IF $n'.count = 0$ % All parents of $n'$ are vis-
            ited.
            THEN Insert $n'$ at the Rear of $Q$.
```

Remarks

1. The above algorithm is similar to breadth-first traversal because the insertion of the children nodes is done at the Rear of Q.
   We could have chosen to do the insertion of the children nodes at the Front of Q. This way the algorithm would have been similar to depth-first traversal.

2. Deadlock situations:
   The algorithm for parents-first traversal never leads to deadlock situations because our graph has no cycles.
   To see why this is the case, let's assume that during the execution of the algorithm, we reach a deadlock situation.
   This means that Q is empty and there is still at least one non-visited node

n. 2, n has never been inserted in Q. Hence, at least one parent (if any) n_1 of n hasn't been visited yet. This means that n_1 has never been inserted in Q either. Hence, at least one parent (if any) n_2 of n_1 hasn't been visited yet. This goes on until we reach a root node r (since the graph is acyclic), and establish that r hasn't been inserted in Q yet. Clearly this is a contradiction, since all root nodes are inserted in Q at the beginning of the algorithm. Hence it is not possible to reach any deadlock situation during the execution of the algorithm.

3. Time Complexity:
   The time complexity of the parents-first traversal is $O(E + N)$.
   Proof:
   It takes $O(N)$ to initialize the counters of the nodes. All the nodes in the graph are inserted in Q and visited exactly once. This takes $O(N)$ time.
   Each time a node n is visited, each one of its children is examined (its counter is updated, and depending of the value of that counter, it may be inserted in Q). Examining a child node takes a constant amount of time. The number of children nodes of n is equal to to number of output edges of n. Hence, since each node in the graph is visited exactly once, this takes $O(E)$ time.

5.2.2 Determining the Notions Used by the Partitioning Algorithm

Determining the CPL

For each node n in the graph, we define *length*(n) to be the length of the longest path from any input node to n, n excluded. Also, for each node n in the graph, we define *pred*(n) to be the predecessor node of n along the longest path from any input node to n, n included (if there is more than one such path, we choose any one of them). Furthermore, for each output node n in the graph, we define exec.path.length(n) to be the length of the longest execution path that ends in n. Finally, we define *cp.last* to be the last node (output node) in the critical path (if there is more than one critical path, we choose anyone of them).

The idea here is to use a parents-first traversal of the graph to determine $length(n)$ for each node n, and $pred(n)$ for each non-root node n in the graph. Then, we determine exec.path.length(n) for each output node n. Finally, we choose the leaf node l such that exec.path.length(l) is the largest among all leaf nodes. The CPL is equal to exec.path.length(l). Clearly, *cp.last* is l. To find the critical path P_{crit}, we use the function *pred*. l is the last node in P_{crit}, $l_1 = pred(l)$ is the node preceding l in P_{crit}, $l_2 = pred(l_1)$ is the node preceding l_1 in P_{crit}, etc., until we reach an input node.
The algorithm is as follows:

```
Let $Q$ be a queue (could be implemented as a linked list).
$Q \leftarrow \emptyset$.
Insert all root nodes in $Q$ (in any order).
FOR all non-root nodes $n$ in the graph DO

    % Initialize the counters of the nodes.
    $n.count \leftarrow$ Number of parent nodes of $n$

FOR all nodes $n$ in the graph DO

    % Initialize $length(n)$ for all nodes $n$ in the graph.
    $length(n) \leftarrow 0$

Repeat until $Q = \emptyset$

    $n \leftarrow$ Front of $Q$.
    Delete $n$ from $Q$.
    visit($n$) % Node $n$ is visited here.
    $n$ is marked visited. % This marking may not be needed.
    IF $n$ is not a leaf node THEN

        FOR all children nodes $n'$ of $n$ DO
            $n'.count \leftarrow n'.count - 1$ % One more parent
            visited.
            $temp \leftarrow length(n) + comp(n) + comm(n, n')$
            % $temp$ is the longest path from any input
```

```
                node to $n'$
                % ($n'$ excluded) that goes through edge $(n, n')$.
                IF $temp > length(n')$
                THEN
                % $temp$ is the longest path that has been
                traversed so far % from any input node to
                $n'$.
                   $length(n') \leftarrow temp$
                   $pred(n') \leftarrow n$
                IF $n'.count = 0$ % All parents of $n'$ are vis-
                ited.
                THEN
                   Insert $n'$ at the Rear of $Q$

% Find CPL
CPL $\leftarrow$ 0
FOR all output nodes $n$ DO

      exec.path.length($n$) $\leftarrow length(n) + comp(n)$
      IF exec.path.length($n$) $>$ CPL
      THEN
            CPL $\leftarrow$ exec.path.length($n$)
            $cp.last \leftarrow n$
```

Time Complexity:

The time complexity to find the CPL and the critical path of a DAG is $O(E + N)$.

Proof:

It takes $O(N)$ to initialize the counters of the nodes. It takes $O(N)$ to initialize $length(n)$ for all nodes n.
Each node in the graph is inserted in Q and visited exactly once. This takes $O(N)$ time.

Each time a node n is visited, each one of its children n' is examined (its counter is updated and depending of the value of that counter it may be inserted in Q, variable *temp* is calculated and depending on its value $length(n')$ and $pred(n')$ could be updated). Examining a child node takes a constant amount of time. The number of children nodes of n is equal to to number of output edges of n. Hence, since each node in the graph is visited exactly once, this takes $O(E)$ time. To determine the CPL and critical path of the graph (once $length(n)$ and $pred(n)$ for all nodes n has been determined), each leaf node l is examined (exec.path.length(l) is calculated, and depending on its value CPL and *cp.last* could be updated). Examining l takes a constant amount of time. Hence this takes $O(N)$ time at the most. Finally, starting from *cp.last* and tracing back along the critical path until we reach an input node takes at the most $O(N)$ time.

Perfect Edges

Consider an edge $e = (n_1, n_2)$.
From a previous lemma, we know that if n_1 has more than one output edge or n_2 has more that one input edge, then e is not a perfect edge. This check can be done in constant ($O(1)$) time.

However, if n_1 has exactly one output edge and n_2 has exactly one input edge, then e could be either perfect or imperfect. In this case, we do a special kind of graph traversal to determine whether the edge is perfect or imperfect. The way we traverse the graph is as follows:

We do a complete graph traversal (e.g. breadth-first or depth-first) in the usual way with the following exception: when node n_1 is visited, its child n_2 is not inserted in the queue Q. If any leaf node is visited, then edge e is not perfect. Otherwise (if no leaf node is visited), edge e is perfect. Hence, whenever a node is visited, we check whether it is a leaf node or not. If it is, we can stop the search immediately and conclude that e is not a perfect edge. If after the search is over none of the nodes visited is a leaf node, then e is a perfect edge.

The idea behind the above procedure is to traverse the graph without going through edge e. By not inserting n_2 in the queue Q, we don't traverse edge e. If a leaf node is reached, then there must exist at least one path from an input node to an output node which does not go through e. This means that there much exist at least one execution path which doesn't go through e. If none of the leaf nodes is reached, then there cannot be any path from an input node to an output node which doesn't go through e. This means that all execution paths must go through e.

Time Complexity:

> The graph traversal described above takes at the most $O(E+N)$ time.
> Therefore, we need $O(E+N)$ time to determine whether an edge is perfect or imperfect.

Remark: The above procedure can be used to check whether edge e is perfect or not even when n_1 has more than one output edge or n_2 has more that one input edge.

Safe Edges

An edge $e = (n_1, n_2)$ is safe if its merger does not result in a cycle. For this to be true, there should not be a path from n_1 to n_2 other than (n_1, n_2) before the merger.
The procedure here is almost the same as the one for perfect edges described above and is as follows:

> We do a graph traversal (e.g. breadth-first or depth-first) starting from node n_1 (initially the queue Q has only node n_1, instead of the root nodes). After n_1 is visited, all of its children nodes are inserted in Q except for node n_2. This way, we traverse all paths emanating from n_1 and which don't go through edge e. If node n_2 is visited, then we can stop the traversal immediately and conclude that e is not a safe edge (i.e. there much exist at least one path from n_1 to n_2 other than

$(n_1, n_2))$. If after all the traversal is complete node n_2 is not visited, then we can conclude that e is a safe edge (i.e. there cannot be a path from n_1 to n_2 other than $(n_1, n_2))$.

Time Complexity:

In the worst case, we will traverse all the graph (except for edge e). Therefore, the above procedure takes at most $O(E + N)$ time.

Risky Edges

From a previous lemma, we know that an edge $e = (n_1, n_2)$ is not risky *if and only if* n_1 has only one output edge and n_2 has only one input edge. This check can be done in constant time. Hence we need $O(1)$ time to find out whether an edge is risky or not.

Determining DepSet(n)

Let n be a node in the DAG.
First, we do a traversal of the graph starting from node n (initially the queue Q has only node n, instead of the root nodes). This will give us all nodes n' such that there is a path from n to n'. Second, we do a backwards traversal of the graph starting from node n (we follow the opposite direction of the edges). This will give us all nodes n' such that there is a path from n' to n.
Initially, we set $DepSet(n)$ to $\emptyset$. Each time we visit a node (other than n), we add it to $DepSet(n)$.
In the worst case, this takes $O(E + N)$ time (complete graph traversal).

Determining ParSet(n)

Let n be a node in the DAG.
One way to determine $ParSet(n)$ is to first determine $DepSet(n)$. By doing that, all nodes n' in the DAG such that n and n' are dependent are marked *visited*. Initially, we set $ParSet(n)$ to $\emptyset$. Then we do a complete traversal of the graph, and any node which was not marked *visited* from the traversal to determine

$DepSet(n)$ is added to $ParSet(n)$. Note that we have to distinguish between the nodes that are marked *visited* during the graph traversal to determine $DepSet(n)$, and during the complete graph traversal to determine $ParSet(n)$. This can be done easily by using different markings (for instance, when we are determining $DepSet(n)$ nodes that are visited are marked with the letter 'D', and when we are determining $ParSet(n)$ nodes that are visited are marked with the letter 'P').
It takes $O(E+N)$ time to determine $DepSet(n)$. Then it takes $O(E+N)$ to do the complete graph traversal to determine $ParSet(n)$. Hence, it takes $O(E+N)$ time complexity to determine $ParSet(n)$.

5.2.3 Time Complexity Using Heuristic 1

Each iteration of the partitioning algorithm consists of choosing the edge to be merged using some heuristic, then the edge chosen is merged[2].
In what follows, we determine the cost for each step of a merging iteration using heuristic 1.

Step 1: First we determine the critical path and the CPL of the task graph[3]. Then for each edge e in the critical path, we check whether e is a safe edge. For all safe edges s found, we check whether s is a perfect edge. Finally, among all edges that are found to be both safe and perfect (if any), we choose the heaviest one.
It takes $O(E+N)$ time to determine the critical path. This path has at the most E edges. For each edge e in the critical path, it takes $O(E+N)$ time to determine whether e is safe and perfect or not. Given the m edges that are found to be perfect and safe $(0 \leq m \leq E)$, it takes $O(m)$ time at the most to determine the heaviest one.
Hence the total time complexity for step 1 is $O(E(E+N))$.

Step 2: The critical path and all m $(0 \leq m \leq E)$ safe edges in it are determined in step 1. Among these safe edges, we determine the ones that are not risky. This takes $O(m)$ time. Among the m' $(0 \leq m' \leq m)$ non-risky edges found,

[2]This is called a merging iteration.
[3]These are actually determined before we start the current merging iteration.

we determine the heaviest one. This takes at the most $O(m')$ time. Hence, it takes at the most $O(E)$ time for step 2.

Step 3: If the DAG is such that each node has at most one output edge, then using a previous corollary regarding risky edges we conclude that step 2 and step 3 are exactly the same, and therefore step 3 is skipped. Otherwise, we do the following.
The critical path and all m $(0 \leq m \leq E)$ safe edges in it are determined in step 1. Among these safe edges $e = (n_1, n_2)$, we determine the ones such that $ParSet(n_1) = ParSet(n_2)$. Therefore for all nodes n that belong to such edges, we need to determine $ParSet(n)$. Since there are at most N nodes along the critical path, this takes at the most $O(N(E+N))$ time. Given 2 sets S_1 and S_2 that have m_1 and m_2 elements respectively, it takes at the most $O(m_1 m_2)$ time to find out whether $S_1 = S_2$. $ParSet(n_1)$ and $ParSet(n_2)$ have at most N elements each. Hence it takes at the most $O(N^2)$ time to find out whether $ParSet(n_1) = ParSet(n_2)$. This check has to be done for all the m safe edges. Hence the total time this takes cannot be more than $O(E.N^2)$. Determining the heaviest edge among all edges found (if any) cannot cost more than $O(E)$.
Therefore, step 3 takes $O(E.N^2)$ time complexity.

Step 4: The critical path and all m $(0 \leq m \leq E)$ safe edges in it are determined in step 1. Also, for each safe edge $e = (n_1, n_2)$, we determined $ParSet(n_1)$ and $ParSet(n_2)$ in Step 3, and we need to determine $ParSet(n_{1,2}) = ParSet(n_1) \cap ParSet(n_2)$. Given 2 sets S_1 and S_2 that have m_1 and m_2 elements respectively, it takes $O(m_1 m_2)$ time at the most to determine $S_1 \cap S_2$. Since $ParSet(n_1)$ and $ParSet(n_2)$ have at most N elements each, it takes at the most $O(N^2)$ time to compute $ParSet(n_{1,2})$. This computation has to be done for each one of the m safe edges found. This takes at the most $O(E.N^2)$ time.
Therefore, it takes at the most $O(E.N^2)$ time to execute step 4.

Step 5: As we saw before, it takes at the most $O(N)$ time to explicitly merge 2 tasks.

Conclusion:

A merging iteration using heuristic 1 costs at the most $O(E(E+N^2))$.
Since there are $N-1$ merging iterations in the partitioning algorithm, the total cost of the partitioning algorithm is $O(EN(E+N^2))$.

Relationship between E and N:

Let E and N be the total number of edges and nodes respectively in a DAG g.
E is equal to the sum of the number of output edges of all nodes n in g[4].
$E = \sum_{n \in g}(\text{number of output edges of } n)$.
For each node n in g, the number m of output edges of n is such that $0 \leq m \leq N-1$[5]. Therefore $0 \leq E \leq N(N-1)$. $E=0$ is the case when all nodes are output nodes. $E = N(N-1)$ is the case when there is an edge from each node n in g to all other nodes in g. These 2 cases never occur in practice. In fact, the case when $E = N(N-1)$ doesn't occur even in theory, since the graph is acyclic[6].

Another expression for Time Complexity:

Since $E < N^2$, the time complexity of the partitioning algorithm using heuristic 1 can be written[7] as $O(E.N^3)$.

Over-Estimation of Time Complexity:

In the previous analysis, we over-estimated the time complexity of the partitioning algorithm because we had to assume the worst case scenario.

[4]We assume that the input nodes don't have any input edges.

[5]$m = 0$ is the case when n is an output node. $m = N-1$ is the case when there is an edge from n to each other node in the graph.

[6]If there is an edge from each node n in g to all other nodes in g, then clearly the graph will have cycles.

[7]Since in this case $O(E+N^2)$ is the same as $O(N^2)$.

For instance, we used E and N for the numbers of edges and nodes respectively along the critical path. Also we used N for the number of elements in $ParSet(n)$ for various nodes n. In addition, we used E for the number of safe edges along the critical path. Clearly for real applications, the actual numbers are usually much smaller than that. Finally as was mentioned before, merging 2 tasks takes $O(N)$ time in the worst case, but it takes a constant amount of time in the average case (for real applications).

Average Time Complexity:

Assume:
Average number of nodes along critical path: constant.
Average number of edges along critical path: constant.
Average number of elements in ParSet(n): constant.
Then:
Step 1: $O(E + N)$.
Step 3: $O(E + N)$.
Step 4: $O(E + N)$.
Hence, the average time complexity of the partitioning algorithm using heuristic 1 is $O(N(E + N))$.

Remark

It is very difficult to determine the average number of edges and nodes along the critical path. These numbers do not necessarily depend on E and N. For instance, we could have a DAG with a very large number of nodes that has a short critical path (i.e. a DAG with a large width), and a DAG with a much smaller number of nodes that has a longer critical path (i.e. DAG with a small width).

Chapter 6

Performance Analysis

6.1 Partitioning Fork and Join DAGs

Since a DAG is composed of fork and join components (Tao Yang: [18, 58, 59]), we study the performance of our partitioning algorithm on these primitive structures to further understand its behavior.

6.1.1 Fork DAGs

Consider the fork DAG shown in figure 6.1. Each c_i is the communication cost of edge (r, n_i), and each e_i is the execution cost of node n_i. Also, e is the execution cost of the root node r.
Without loss of generality, assume that the leaf nodes are sorted such that $c_i + e_i \geq c_{i+1} + e_{i+1}$, $1 \leq i \leq m-1$.

Optimal Partition

The critical path of the fork DAG shown in figure 6.1 is (r, n_1). Hence initially, the CPL is $l_0 = e + c_1 + e_1$. Clearly, the CPL of the optimal task graph is $l_{opt} \leq l_0$.

The main thing to notice here is that whenever an edge (r, n_i) is merged, all the other execution paths (r, n_j) $(j \neq i)$ are increased in length by e_i.

If $l_{opt} = l_0$ then the initial task graph is the optimal one.
If $l_{opt} < l_0$ then in the optimal partition, nodes r and n_1 have to belong to the same task. Otherwise, l_{opt} cannot be smaller than l_0. Hence, edge (r, n_1) has to

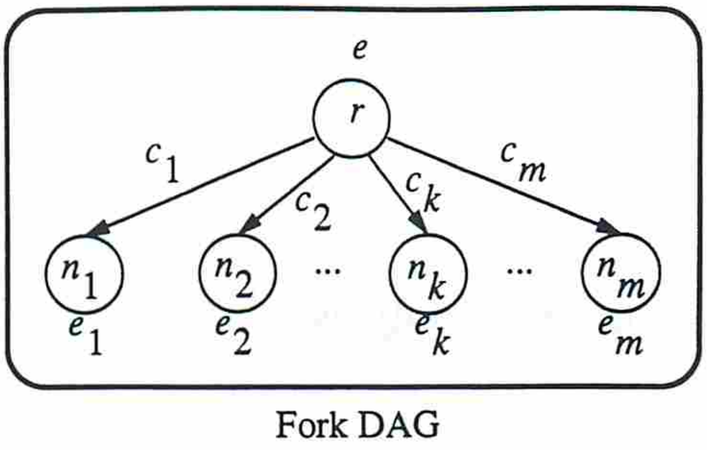
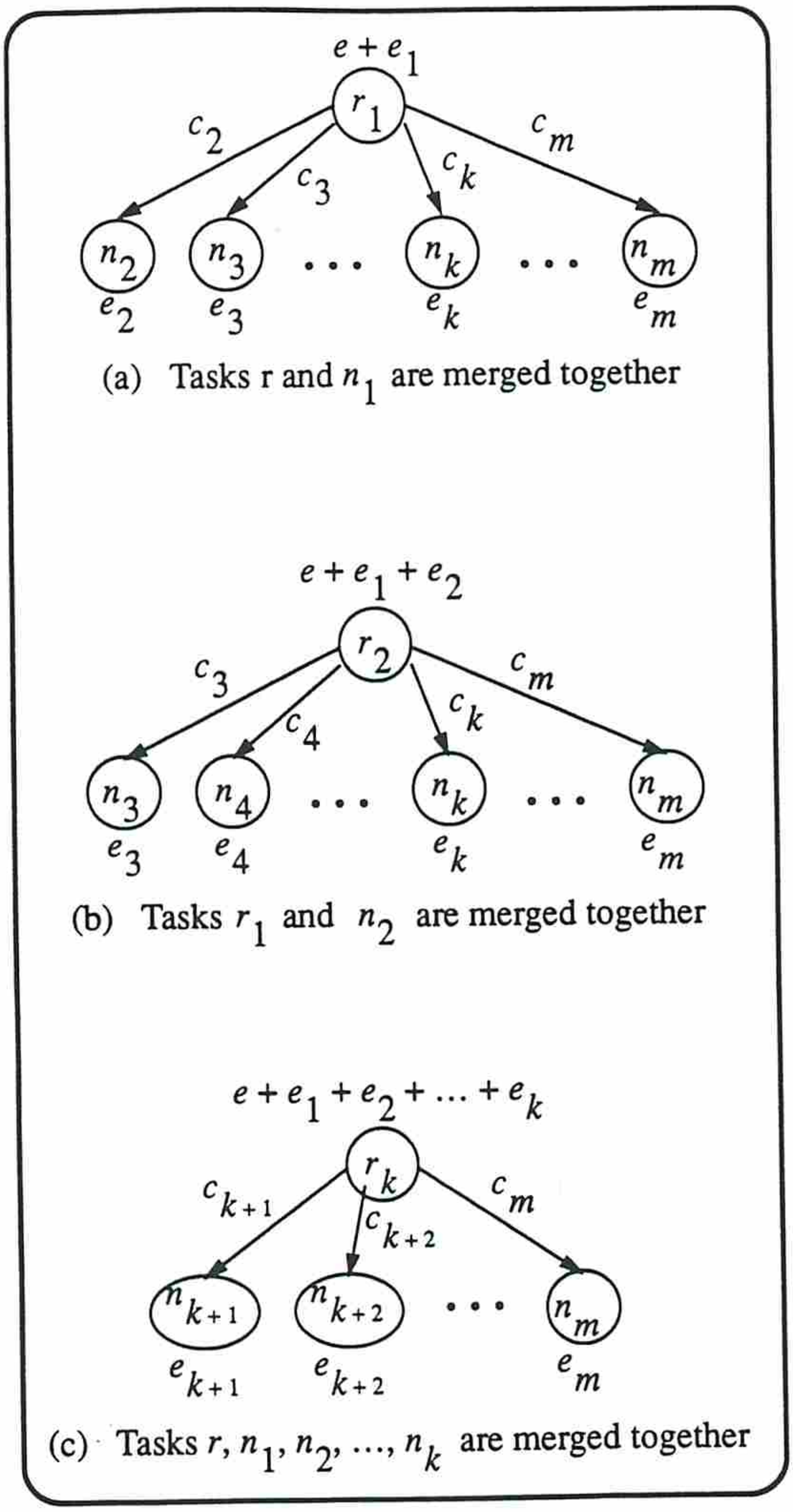


Figure 6.1: Fork DAG

be merged. The result of this merger is shown in figure 6.2 (a). The new critical path is (r_1, n_2), and the new CPL is $l_1 = e + e_1 + c_2 + e_2$.
Again, $l_{opt} \leq l_1$.
If $l_{opt} = l_1$ then the task graph in figure 6.2 (a) is the optimal one.
If $l_{opt} < l_1$ then in the optimal partition, the root node r_1 and node n_2 have to belong to the same task. Otherwise, l_{opt} cannot be smaller than l_1. Hence, edge (r_1, n_2) has to be merged. The result of this merger is shown in figure 6.2 (b). The new critical path is (r_2, n_3), and the new CPL is $l_2 = e + e_1 + e_2 + c_3 + e_3$.
This process goes on, and after the k'th merging step, the task graph is shown in figure 6.2 (c). The critical path of this task graph is (r_k, n_{k+1}), and its CPL is $l_k = e + e_1 + e_2 + \cdots + e_k + c_{k+1} + e_{k+1}$.
This process could go on until all tasks are merged together and the task graph is constituted of a single task. The CPL in this case is $l_m = e + e_1 + e_2 + \cdots + e_m$.
Note that the order of merging the edges in the graph does not matter, and we always get to the same result. In other words, to obtain task $\{r, n_1, n_2, \ldots, n_k\}$, we need to merge the k edges whose costs are $c_1, c_2, \ldots, c_k$ in any order.

The above intuitive analysis leads us to the following theorem.

Theorem: The optimal partition for the fork DAG is constituted of the following tasks:



(a) Tasks r and n_1 are merged together

(b) Tasks r_1 and n_2 are merged together

(c) Tasks $r, n_1, n_2, \dots, n_k$ are merged together

Figure 6.2: Determining optimal partition for fork DAGs

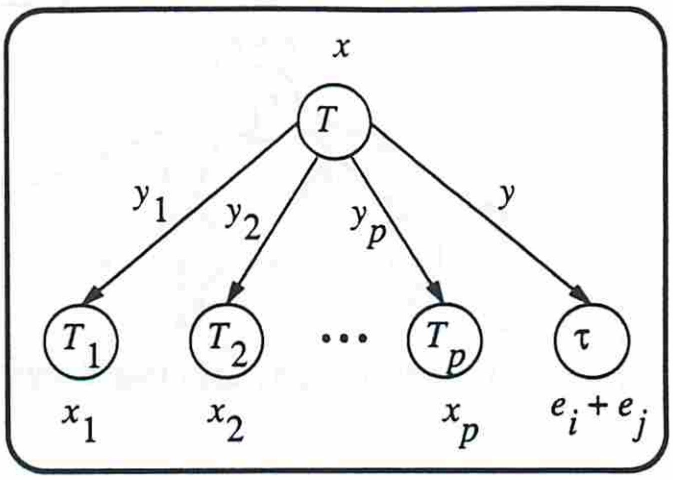


Figure 6.3: Proof: optimal partition for fork DAGs

$\{r, n_1, n_2, \ldots, n_k\}$, $\{n_{k+1}\}$, $\{n_{k+2}\}$, ..., $\{n_m\}$, where $0 \leq k \leq m$.
For $k = 0$ the optimal partition is the trivial partition, and for $k = m$ the optimal partition is the singleton partition.

Proof

First, let's prove that the optimal partition is constituted of the following tasks: $\{r, N_1, N_2, \ldots, N_k\}$, $\{N_{k+1}\}$, $\{N_{k+2}\}$, ..., $\{N_m\}$, where $0 \leq k \leq m$ and N_i = some n_j (i not necessarily equal to j).
In other words, in the optimal partition, there is a task containing r and zero or more other n_i's, and the rest of the n_i's are in separate tasks (i.e. these tasks are constituted of a single n_i).
This is also equivalent to saying that in the optimal partition, any task that doesn't contain r cannot contain more than a single n_i.

Intuitively, since merging 2 independent tasks together doesn't reduce communication cost, it will never decrease the parallel execution time (CPL), and therefore, any task in the optimal partition should not consist entirely of n_i's. Hence, any task which does not contain r is constituted of a single n_i.

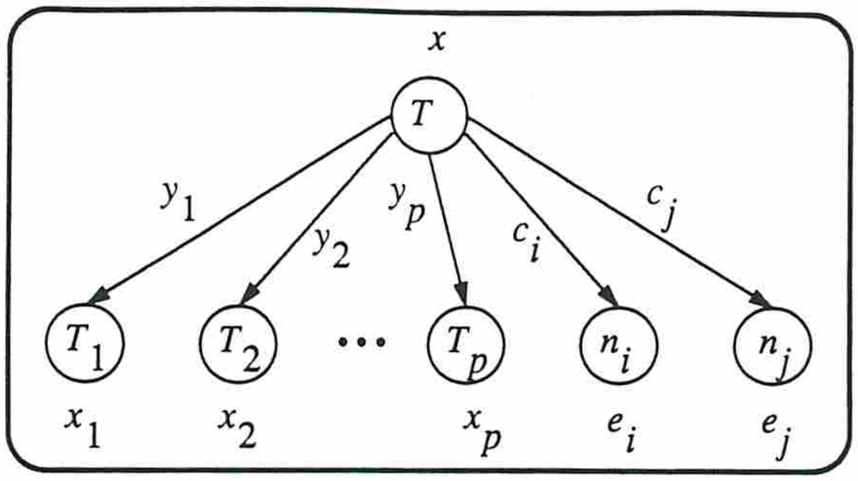


Figure 6.4: Proof: optimal partition for fork DAGs

To prove this more formally, let's show that any partition Π that has a task τ not containing r and that contains more than one n_i has a CPL that is greater or equal than the one of the partition obtained from Π by putting each n_i that belongs to τ in a separate task.
Without any loss of generality, let $\tau = \{n_i, n_j\}$ $(i < j)$. The task graph of Π is shown in figure 6.3. Clearly, task T contains r[1]. x_k is the execution cost of task T_k. y_k is the communication cost of edge (T, T_k). x is the execution cost of task T. The communication cost of edge (T, τ) is $y = c_i + delay(data(T, \tau))$.
Also without any loss of generality, assume that $y_k + x_k \geq y_{k+1} + x_{k+1}$, $1 \leq k \leq p-1$.
The CPL of Π is
$l = x + max(y_1 + x_1, y + e_i + e_j)$.
Now consider the partition Π' obtained from Π by putting n_i and n_j in separate tasks. The task graph corresponding to Π' is shown in figure 6.4.
The CPL of Π' is
$l' = x + max(y_1 + x_1, c_i + e_i)$.
Since $y + e_i + e_j > c_i + e_i$, then $l \geq l'$.

[1]T could be constituted entirely of r.

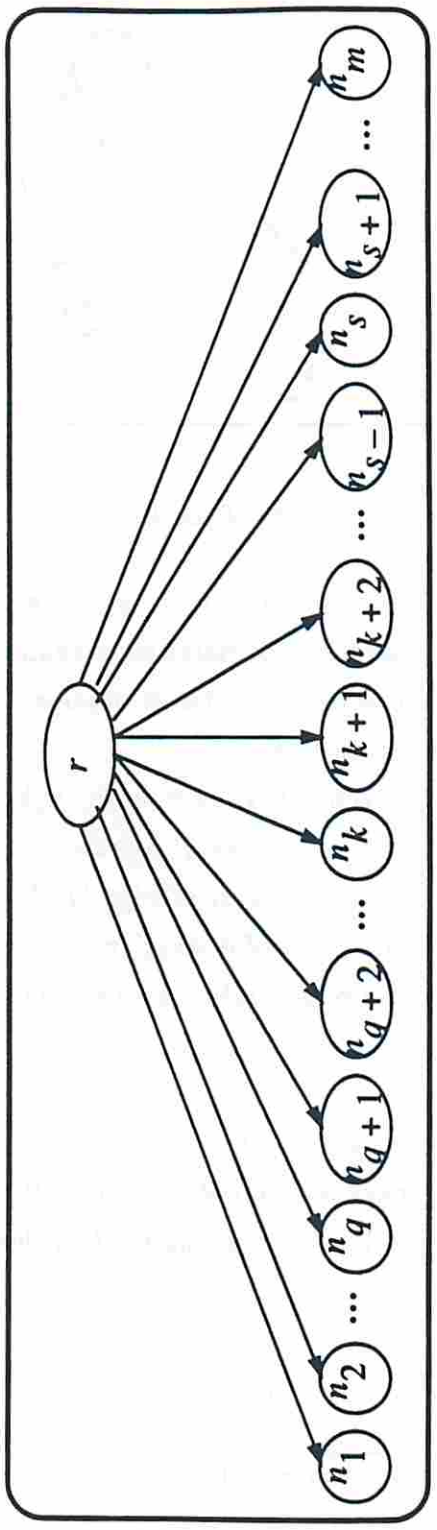


Figure 6.5: Proof: optimal partition for fork DAGs

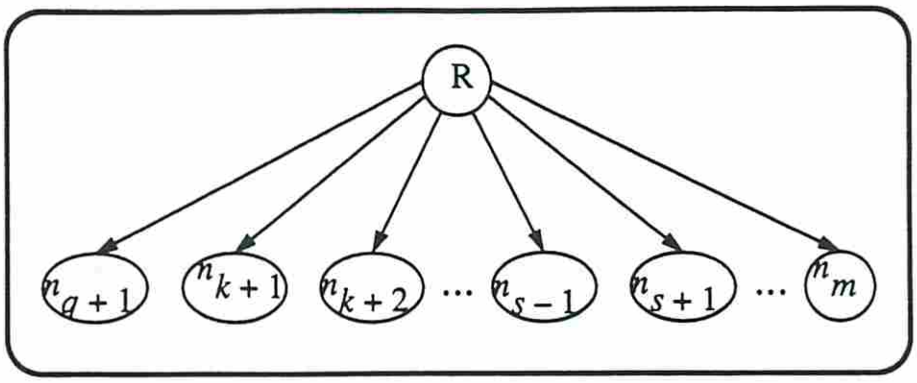


Figure 6.6: Proof: optimal partition for fork DAGs

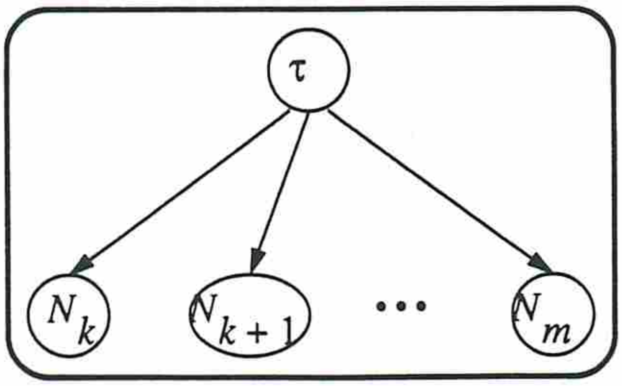


Figure 6.7: Proof: optimal partition for fork DAGs

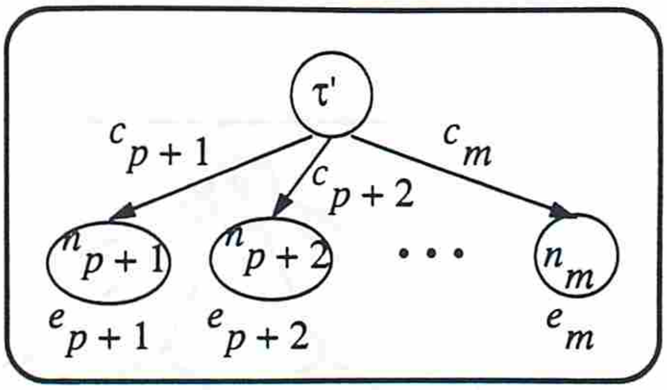


Figure 6.8: Proof: optimal partition for fork DAGs

Now let's prove the claim in the theorem.

First we use an intuitive reasoning.

Let's assume that the claim is not correct.

Hence there exists at least one n_s, $s > k$, such that the optimal partition is constituted of the following tasks:

$R = \{r, n_1, n_2, \ldots, n_q, n_s, n_{q+2}, n_{q+3}, \ldots, n_k\}$, $\{n_{k+1}\}$, $\{n_{k+2}\}$, ... , $\{n_{s-1}\}$, $\{n_{q+1}\}$, $\{n_{s+1}\}$, $\{n_{s+2}\}$, ... , $\{n_m\}$.

Refer to figure 6.5.

The task graph corresponding to this partition is shown in figure 6.6.

The CPL of this optimal partition is

$l_{opt} = e + e_1 + e_2 + \cdots + e_q + e_{q+2} + e_{q+3} + \cdots + e_k + e_s + c_{q+1} + e_{q+1}$.

Consider the partition constituted of the following tasks:

$\{r, n_1, n_2, \ldots, n_q\}$, $\{n_{q+1}\}$, $\{n_{q+2}\}$, ... , $\{n_m\}$.

Its CPL is

$l = e + e_1 + e_2 + \cdots + e_q + c_{q+1} + e_{q+1}$.

Note that $l < l_{opt}$, which should not be. Hence we have a contradiction, and therefore our assumption is not possible.

Then the claim of the theorem is correct.

Now let's prove the claim in the theorem more formally.

Assume that the claim is not correct.

The optimal partition Π_{opt} is constituted of the following tasks:

$\tau = \{r, N_1, N_2, \ldots, N_k\}$, $\{N_{k+1}\}$, $\{N_{k+2}\}$, ..., $\{N_m\}$, where $0 \leq k \leq m$ and $N_i =$ some n_j (i not necessarily equal to j).

Without any loss of generality, assume that $k \geq 1$. When $k = 0$, the optimal partition is the trivial partition, and therefore the claim is true.

Also, without any loss of generality, assume that the N_i's $(1 \leq i \leq k)$ in τ are ordered such that if $N_j = n_{j1}$ and $N_{j+1} = n_{j2}$ then $j2 > j1$, $1 \leq j \leq k-1$.

Let $N_0 = r$ and $n_0 = r$.

Let p be the smallest integer such that $N_p = n_p$ and $N_{p+1} \neq n_{p+1}$, $0 \leq p \leq k-1$. For instance, if $\tau = \{r, n_1, n_2, n_3, n_5, \ldots\}$ then $p = 3$. If $\tau = \{r, n_1, n_5, \ldots\}$ then $p = 1$. If $\tau = \{r, n_2, \ldots\}$ then $p = 0$.

We have $\tau = \{r, n_1, n_2, \ldots, n_p, N_{p+1}, N_{p+2}, \ldots, N_k\}$, $0 \leq p \leq k-1$.

Clearly, $N_i \neq n_{p+1}$, $p+1 \leq i \leq k$. Therefore, n_{p+1} is in a task by itself. The task graph corresponding to Π_{opt} is shown in figure 6.7.

Since one of the leaf nodes is n_{p+1} and all n_i's such that $1 \leq i \leq p$ are in task τ, then critical path of this task graph is (τ, n_{p+1}). Hence the CPL of Π_{opt} is

$l_{opt} = comp(\tau) + comm(\tau, n_{p+1}) + comp(n_{p+1})$.

Let the execution time of N_i be E_i, $1 \leq i \leq m$.

$comp(\tau) = e + e_1 + e_2 + \cdots + e_p + E_{p+1} + E_{p+2} + \cdots + E_k$.

$comm(\tau, n_{p+1}) = c_{p+1}$.

$comp(n_{p+1}) = e_{p+1}$.

Hence

$l_{opt} = e + e_1 + e_2 + \cdots + e_p + E_{p+1} + E_{p+2} + \cdots + E_k + c_{p+1} + e_{p+1}$.

Consider the partition Π constituted of the following tasks:

$\tau' = \{r, n_1, n_2, \ldots, n_p\}$, $\{n_{p+1}\}$, $\{n_{p+2}\}$, ..., $\{n_m\}$.

The task graph corresponding to Π is shown in figure 6.8.

Its critical path is (τ', n_{p+1}) and its CPL is

$l = e + e_1 + e_2 + \cdots + e_p + c_{p+1} + e_{p+1}$.

Note that $l < l_{opt}$, which means that we have a contradiction. Therefore, our assumption is not correct, and the claim of the theorem is correct.

Using Heuristic 1

The critical path of the fork DAG in figure 6.1 is (r, n_1). Hence the first edge chosen to be merged is (r, n_1). The result of this merger is shown in figure 6.2 (a). The new critical path is (r_1, n_2). Hence the next edge to be merged is (r_1, n_2). The resulting task graph is shown in figure 6.2 (b). This process goes on, and at the k'th merging step, edge (r_{k-1}, n_k)[2] is chosen for merger. The result if this merger is shown is figure 6.2 (c). This merging process goes on until we reach the singleton partition.

It is quite clear that using Heuristic 1, the partitions that we get during the iterations of the partitioning algorithm are constituted of the following tasks:
$\{r, n_1, n_2, \ldots, n_k\}$, $\{n_{k+1}\}$, $\{n_{k+2}\}$, …, $\{n_m\}$, where $0 \le k \le m$.
From the above theorem, we conclude that our algorithm always leads to the optimal partition.

Some Analysis

Let Π_k be the partition constituted of the following tasks:
$\{r, n_1, n_2, \ldots, n_k\}$, $\{n_{k+1}\}$, $\{n_{k+2}\}$, …, $\{n_m\}$, where $0 \le k \le m$.
Let l_k be the critical path length of Π_k.
From the above theorem, we know that the optimal partition is one of the Π_k's, $0 \le k \le m$.
Note that using our partitioning algorithm with Heuristic 1, Π_k is the partition obtained after the k'th merging step, and l_k is the CPL of the task graph after the k'th merging step.
The task graph corresponding to partition Π_k is shown in figure 6.2 (c), with r_0 and e_0 defined to be r and e respectively.

$l_k = e + e_1 + e_2 + \cdots + e_k + e_{k+1} + c_{k+1}$, $0 \le k \le m - 1$.
$l_m = e + e_1 + e_2 + \cdots + e_m$.
Note that $l_0 = e + e_1 + c_1$.
$l_k - l_{k-1} = e_{k+1} + c_{k+1} - c_k$, $1 \le k \le m - 1$.

[2]Assume that r_0 is defined to be r, so that when $k = 1$ $r_{k-1} = r_0 = r$.

Hence, for $1 \leq k \leq m-1$, we have

$l_k \leq l_{k-1} \Leftrightarrow c_k \geq e_{k+1} + c_{k+1}$

AND

$l_k \geq l_{k-1} \Leftrightarrow c_k \leq e_{k+1} + c_{k+1}$.

$l_k - l_m = c_{k+1} - e_{k+2} - e_{k+3} - \cdots - e_m,\ 0 \leq k \leq m-1$.

Hence, for $0 \leq k \leq m-2$, we have

$l_k \leq l_m \Leftrightarrow c_{k+1} \leq e_{k+2} + e_{k+3} + \cdots + e_m$.

Note that $l_{m-1} - l_m = c_m$. Hence

$l_{m-1} > l_m$, and therefore Π_{m-1} can never be the optimal partition.

Corollary: Assume that there exists an integer q, $1 \leq q \leq m-2$, such that

$\forall\ k,\ 1 \leq k \leq q,\ c_k \geq e_{k+1} + c_{k+1}$

AND

$\forall\ k,\ q+1 \leq k \leq m-1,\ c_k \leq e_{k+1} + c_{k+1}$

AND

$c_{q+1} \leq e_{q+2} + e_{q+3} + \cdots + e_m$.

Then Π_q is the optimal partition.

Proof

Using the analysis above, we obtain the following:

$\forall\ k,\ 1 \leq k \leq q,\ l_k \leq l_{k-1}$

AND

$\forall\ k,\ q+1 \leq k \leq m-1,\ l_k \geq l_{k-1}$

AND

$l_q \leq l_m$.

Hence, $l_q \leq l_k,\ 0 \leq k \leq m$.

Therefore, Π_q is the optimal partition.

Examples

1. Consider the fork DAG shown in figure 6.9 (a).
   Using the above corollary with $q = 2$, we conclude that Π_2 is the optimal partition.

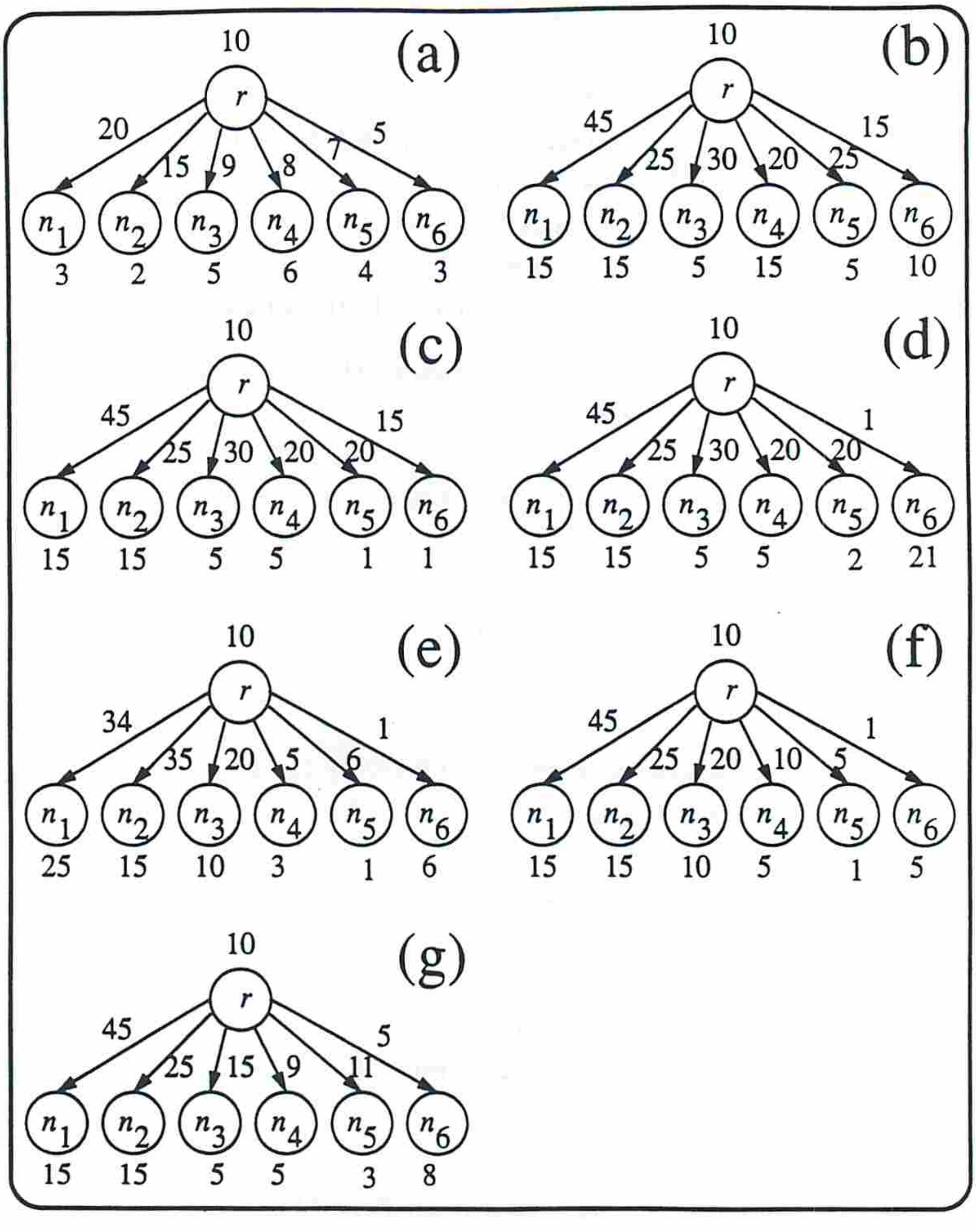


Figure 6.9: Examples of fork DAGs

2. Consider the fork DAG shown in figure 6.9 (b).
   Using the above corollary with $q = 1$, we conclude that Π_1 is the optimal partition.

3. Consider the fork DAG shown in figure 6.9 (c).
   Here we cannot apply the above corollary because q does not exist.
   $l_0 = 70$, $l_1 = 65$, $l_2 = 75$, $l_3 = 70$, $l_4 = 71$, $l_5 = 67$, $l_6 = 52$.
   Therefore, Π_6 is the optimal partition.

4. Consider the fork DAG shown in figure 6.9 (d).
   Here we cannot apply the above corollary because q does not exist.
   $l_0 = 70$, $l_1 = 65$, $l_2 = 75$, $l_3 = 70$, $l_4 = 72$, $l_5 = 74$, $l_6 = 73$.
   Therefore, Π_1 is the optimal partition.

5. Consider the fork DAG shown in figure 6.9 (e).
   Here we cannot apply the above corollary because q does not exist.
   $l_0 = 69$, $l_1 = 85$, $l_2 = 80$, $l_3 = 68$, $l_4 = 70$, $l_5 = 71$, $l_6 = 70$.
   Therefore, Π_3 is the optimal partition.

6. Consider the fork DAG shown in figure 6.9 (f).
   Here we cannot apply the above corollary because q does not exist.
   $l_0 = 70$, $l_1 = 65$, $l_2 = 70$, $l_3 = 65$, $l_4 = 61$, $l_5 = 62$, $l_6 = 61$.
   Therefore, Π_4 and Π_6 are the optimal partitions.

7. Consider the fork DAG shown in figure 6.9 (g).
   Using the above corollary with $q = 3$, we conclude that Π_3 is the optimal partition.

Using Sarkar's Partitioning Method

Examples

1. Consider the fork DAG shown in figure 6.9 (a).
   The edges in the graph sorted in decreasing communication cost are as follows:
   (r, n_1), (r, n_2), (r, n_3), (r, n_4), (r, n_5), (r, n_6).

Initially, the CPL of the graph is 33. Merging edge (r, n_1) results in a CPL of 30, and therefore it is accepted. Merging edge (r, n_2) results in a CPL of 29, and therefore it is accepted. Merging edge (r, n_3) results in a CPL of 34, and therefore it is not accepted. Merging edge (r, n_4) results in a CPL of 35, and therefore it is not accepted. Merging edge (r, n_5) results in a CPL of 33, and therefore it is not accepted. Finally, merging edge (r, n_6) results in a CPL of 32, and therefore it is not accepted.
Hence, the partition obtained using Sarkar's method is Π_2, which is as was seen earlier the optimal partition.

2. Consider the fork DAG shown in figure 6.9 (b).
   The edges in the graph sorted in decreasing communication cost are as follows:
   (r, n_1), (r, n_3), (r, n_2), (r, n_5), (r, n_4), (r, n_6).
   Initially, the CPL of the graph is 70. Merging edge (r, n_1) results in a CPL of 65, and therefore it is accepted. Merging edge (r, n_3) results in a CPL of 70, and therefore it is not accepted. Merging edge (r, n_2) results in a CPL of 75, and therefore it is not accepted. Merging edge (r, n_5) results in a CPL of 70, and therefore it is not accepted. Merging edge (r, n_4) results in a CPL of 80, and therefore it is not accepted. Finally, merging edge (r, n_6) results in a CPL of 75, and therefore it is not accepted.
   Hence, the partition obtained using Sarkar's method is Π_1, which is as was seen earlier the optimal partition.

3. Consider the fork DAG shown in figure 6.9 (c).
   The edges in the graph sorted in decreasing communication cost are as follows:
   (r, n_1), (r, n_3), (r, n_2), (r, n_4), (r, n_5), (r, n_6).
   Initially, the CPL of the graph is 70. Merging edge (r, n_1) results in a CPL of 65, and therefore it is accepted. Merging edge (r, n_3) results in a CPL of 70, and therefore it is not accepted. Merging edge (r, n_2) results in a CPL of 75, and therefore it is not accepted. Merging edge (r, n_4) results in a CPL of 70, and therefore it is not accepted. Merging edge (r, n_5) results in a CPL

of 66, and therefore it is not accepted. Finally, merging edge (r, n_6) results in a CPL of 66, and therefore it is not accepted.
Hence, the partition obtained using Sarkar's method is Π_1. As was seen earlier, the optimal partition for this case is Π_6. Hence, Sarkar's method does not lead to the optimal partition.

4. Consider the fork DAG shown in figure 6.9 (e).
The edges in the graph sorted in decreasing communication cost are as follows:
(r, n_2), (r, n_1), (r, n_3), (r, n_5), (r, n_4), (r, n_6).
Initially, the CPL of the graph is 69. Merging edge (r, n_2) results in a CPL of 84, and therefore it is not accepted. Merging edge (r, n_1) results in a CPL of 85, and therefore it is not accepted. Merging edge (r, n_3) results in a CPL of 79, and therefore it is not accepted. Merging edge (r, n_5) results in a CPL of 70, and therefore it is not accepted. Merging edge (r, n_4) results in a CPL of 72, and therefore it is not accepted. Finally, merging edge (r, n_6) results in a CPL of 75, and therefore it is not accepted.
Hence, the partition obtained using Sarkar's method is Π_0. As was seen earlier, the optimal partition for this case is Π_3. Hence, Sarkar's method does not lead to the optimal partition.

5. Consider the fork DAG shown in figure 6.9 (f).
The edges in the graph sorted in decreasing communication cost are as follows:
(r, n_1), (r, n_2), (r, n_3), (r, n_4), (r, n_5), (r, n_6).
Initially, the CPL of the graph is 70. Merging edge (r, n_1) results in a CPL of 65, and therefore it is accepted. Merging edge (r, n_2) results in a CPL of 70, and therefore it is not accepted. Merging edge (r, n_3) results in a CPL of 75, and therefore it is not accepted. Merging edge (r, n_4) results in a CPL of 70, and therefore it is not accepted. Merging edge (r, n_5) results in a CPL of 66, and therefore it is not accepted. Finally, merging edge (r, n_6) results in a CPL of 70, and therefore it is not accepted.
Hence, the partition obtained using Sarkar's method is Π_1. As was seen

earlier, Π_4 and Π_6 are the optimal partitions. Hence, Sarkar's method does not lead to the optimal partition.

6. Consider the fork DAG shown in figure 6.9 (g).
   The edges in the graph sorted in decreasing communication cost are as follows:
   (r, n_1), (r, n_2), (r, n_3), (r, n_5), (r, n_4), (r, n_6).
   Initially, the CPL of the graph is 70. Merging edge (r, n_1) results in a CPL of 65, and therefore it is accepted. Merging edge (r, n_2) results in a CPL of 60, and therefore it is accepted. Merging edge (r, n_3) results in a CPL of 59, and therefore it is accepted. Merging edge (r, n_5) results in a CPL of 62, and therefore it is not accepted. Merging edge (r, n_4) results in a CPL of 64, and therefore it is not accepted. Finally, merging edge (r, n_6) results in a CPL of 67, and therefore it is not accepted.
   Hence, the partition obtained using Sarkar's method is Π_3, which is as was seen earlier the optimal partition.

Theorem: If there exists an integer q, $1 \leq q \leq m-2$, such that
$\forall\, k,\ 1 \leq k \leq q,\ c_k \geq e_{k+1} + c_{k+1}$
AND
$\forall\, k,\ q+1 \leq k \leq m-1,\ c_k \leq e_{k+1} + c_{k+1}$
AND
$c_{q+1} \leq e_{q+2} + e_{q+3} + \cdots + e_m$.
then Sarkar's method finds the optimal partition Π_q.

Proof

Clearly, in this case we have
$\forall\, k,\ 1 \leq k \leq q,\ c_k > c_{k+1}$.
Also, since $c_q \geq e_{q+1} + c_{q+1}$, and we know that $c_i + e_i \geq c_{i+1} + e_{i+1}$, $1 \leq i \leq m-1$, then $c_q \geq c_k + e_k$, $q+1 \leq k \leq m$.
Therefore, $c_q > c_k$, $q+1 \leq k \leq m$.
Therefore, the edges are merged in the following order:
(r, n_1), (r, n_2), …, (r, n_q), ….

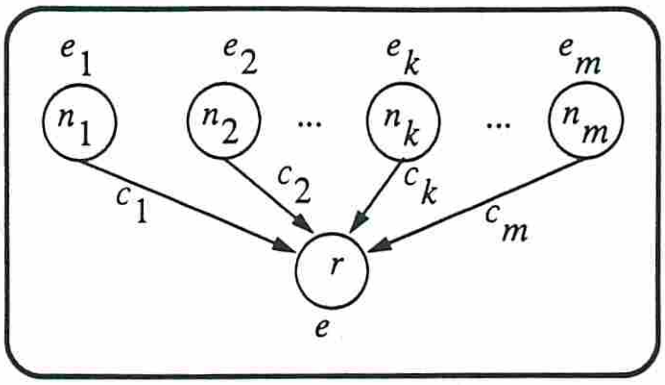


Figure 6.10: Join DAG

The merging of (r, n_1) leads to partition Π_1. Since $l_1 \leq l_0$, this merger is accepted. Next, (r, n_2) is merged and we get partition Π_2. Since $l_2 \leq l_1$, this merger is accepted. Since $l_k \leq l_{k-1}$, $1 \leq k \leq q$, then this process goes on until we reach partition Π_q, which is the optimal partition.

The fork DAGs in figure 6.9 (a), (b) and (g) are examples of such situation.

Theorem: Assume that the optimal partition of the fork DAG is Π_p, $1 \leq p \leq m$, and that Π_i, $0 \leq i \leq p-1$, is not an optimal partition.
If there exists an s, $1 \leq s \leq p$, such that merging edge (r, n_s) is not accepted (i.e. we have an increase in the CPL), then Sarkar's method does not find the optimal partition.

Proof

We know that if Π_i is an optimal partition, then $i \geq p$. If edge (r, n_s) is not merged, then we can never obtain any partition Π_i, $s \leq i \leq m$. Hence, the optimal partition can never be obtained.

The fork DAGs in figure 6.9 (c), (e) and (f) are examples of such situation.

6.1.2 Join DAGs

Consider the join DAG shown in figure 6.10. Each c_i is the communication cost of edge (n_i, r), and each e_i is the execution cost of node n_i. Also, e is the execution cost of the leaf node r.
Without loss of generality, assume that the root nodes are sorted such that $c_i + e_i \geq c_{i+1} + e_{i+1}$, $1 \leq i \leq m-1$.

The case of join DAGs is the same as the one for fork DAGs, and the analysis here is the same as for fork DAGs. We just have to reverse the direction of the edges in the graph.

The critical path of the join DAG shown in figure 6.10 is (n_1, r). Hence initially, the CPL is $l_0 = e + c_1 + e_1$.

Again, the main thing to notice here is that whenever an edge (n_i, r) is merged, all the other execution paths (n_j, r) $(j \neq i)$ are increased in length by e_i.

Theorem: The optimal partition for the join DAG is constituted of the following tasks:
$\{r, n_1, n_2, \ldots, n_k\}$, $\{n_{k+1}\}$, $\{n_{k+2}\}$, …, $\{n_m\}$, where $0 \leq k \leq m$.
For $k = 0$ the optimal partition is the trivial partition, and for $k = m$ the optimal partition is the singleton partition.

Proof: The proof is exactly the same as the one for the theorem for fork DAGs.

Using Heuristic 1

The critical path of the join DAG in figure 6.10 is (n_1, r). Hence the first edge chosen to be merged is (n_1, r). The result of this merger is shown in figure 6.11 (a). The new critical path is (n_2, r_1). Hence the next edge to be merged is (n_2, r_1). The resulting task graph is shown in figure 6.11 (b). This process goes on, and at the k'th merging step, edge (n_k, r_{k-1})[3] is chosen for merger. The result if this merger is shown is figure 6.11 (c). This merging process goes on until we reach the singleton partition.

[3]Assume that r_0 is defined to be r, so that when $k = 1$ $r_{k-1} = r_0 = r$.

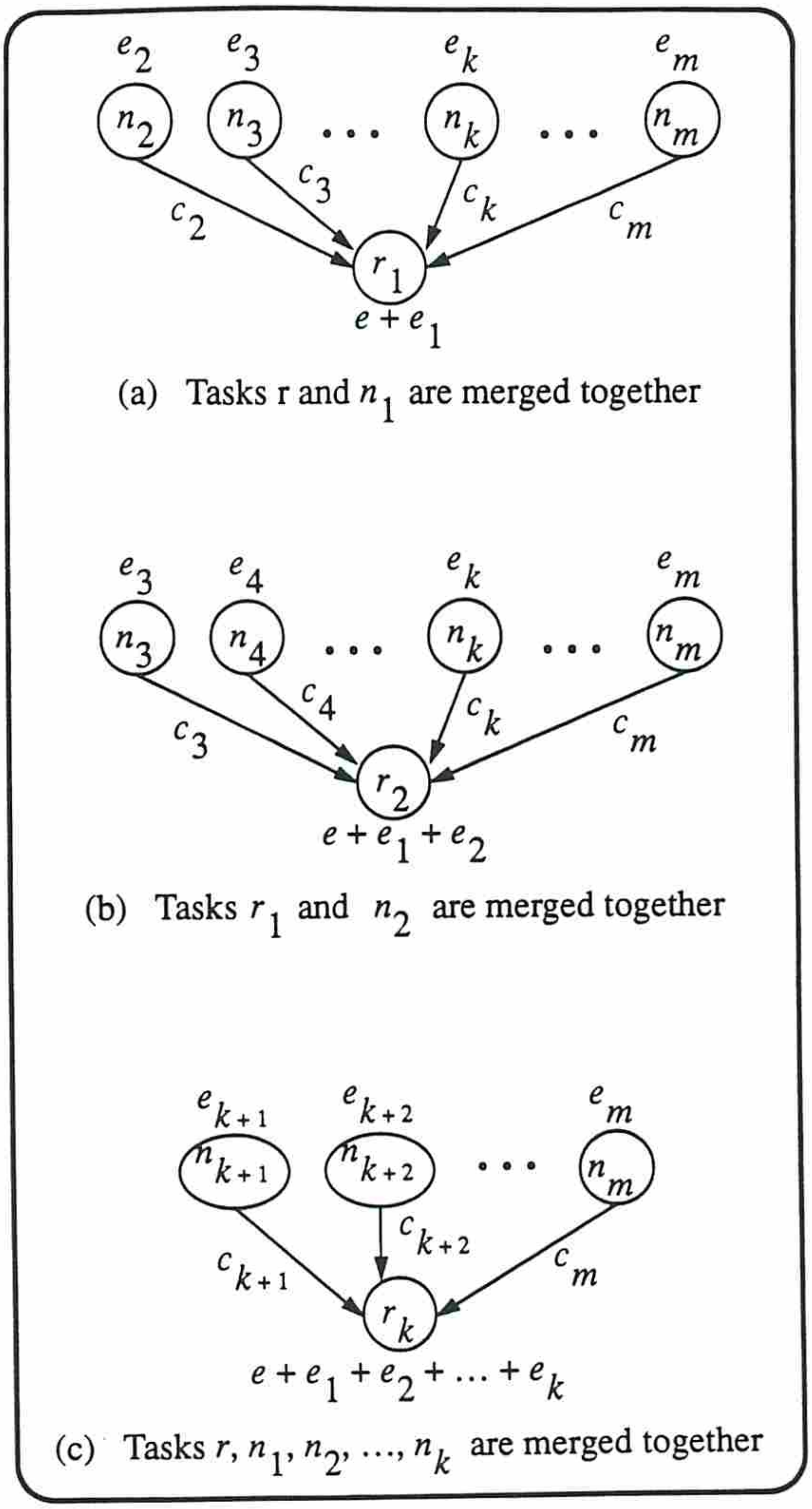

e_2 e_3 e_k e_m
n_2 n_3 ... n_k ... n_m
c_2 c_3 c_k c_m
r_1
$e + e_1$
(a) Tasks r and n_1 are merged together
e_3 e_4 e_k e_m
n_3 n_4 ... n_k ... n_m
c_3 c_4 c_k c_m
r_2
$e + e_1 + e_2$
(b) Tasks r_1 and n_2 are merged together
e_{k+1} e_{k+2} e_m
n_{k+1} n_{k+2} ... n_m
c_{k+1} c_{k+2} c_m
r_k
$e + e_1 + e_2 + \ldots + e_k$
(c) Tasks $r, n_1, n_2, \ldots, n_k$ are merged together

Figure 6.11: Merging steps for join DAGs using our heuristic

It is quite clear that using Heuristic 1, the partitions that we get during the iterations of the partitioning algorithm are constituted of the following tasks: $\{r, n_1, n_2, \ldots, n_k\}$, $\{n_{k+1}\}$, $\{n_{k+2}\}$, ..., $\{n_m\}$, where $0 \leq k \leq m$.
From the above theorem, we conclude that our algorithm always leads to the optimal partition.

Some Analysis

Let Π_k be the partition constituted of the following tasks:
$\{r, n_1, n_2, \ldots, n_k\}$, $\{n_{k+1}\}$, $\{n_{k+2}\}$, ..., $\{n_m\}$, where $0 \leq k \leq m$.
Let l_k be the critical path length of Π_k.
From the above theorem, we know that the optimal partition is one of the Π_k's, $0 \leq k \leq m$.
Note that using our partitioning algorithm with Heuristic 1, Π_k is the partition obtained after the k'th merging step, and l_k is the CPL of the task graph after the k'th merging step.
The task graph corresponding to partition Π_k is shown in figure 6.11 (c), with r_0 and e_0 defined to be r and e respectively.

$l_k = e + e_1 + e_2 + \cdots + e_k + e_{k+1} + c_{k+1}$, $0 \leq k \leq m-1$.
$l_m = e + e_1 + e_2 + \cdots + e_m$.
Note that $l_0 = e + e_1 + c_1$.
$l_k - l_{k-1} = e_{k+1} + c_{k+1} - c_k$, $1 \leq k \leq m-1$.
Hence, for $1 \leq k \leq m-1$, we have
$l_k \leq l_{k-1} \Leftrightarrow c_k \geq e_{k+1} + c_{k+1}$
AND
$l_k \geq l_{k-1} \Leftrightarrow c_k \leq e_{k+1} + c_{k+1}$.
$l_k - l_m = c_{k+1} - e_{k+2} - e_{k+3} - \cdots - e_m$, $0 \leq k \leq m-1$.
Hence, for $0 \leq k \leq m-2$, we have
$l_k \leq l_m \Leftrightarrow c_{k+1} \leq e_{k+2} + e_{k+3} + \cdots + e_m$.
Note that $l_{m-1} - l_m = c_m$. Hence
$l_{m-1} > l_m$, and therefore Π_{m-1} can never be the optimal partition.

Corollary: Assume that there exists an integer q, $1 \leq q \leq m-2$, such that

$\forall\ k,\ 1 \leq k \leq q,\ c_k \geq e_{k+1} + c_{k+1}$

AND

$\forall\ k,\ q+1 \leq k \leq m-1,\ c_k \leq e_{k+1} + c_{k+1}$

AND

$c_{q+1} \leq e_{q+2} + e_{q+3} + \cdots + e_m.$

Then Π_q is the optimal partition.

Proof: The proof is exactly the same as for the corollary for fork DAGs.

Examples: The same examples that were used for fork DAGs can be used here. We just have to reverse the direction of the edges in the graph.

Using Sarkar's Partitioning Method

The case for join DAGs is the same as the one for fork DAGs.

Theorem: If there exists an integer q, $1 \leq q \leq m-2$, such that

$\forall\ k,\ 1 \leq k \leq q,\ c_k \geq e_{k+1} + c_{k+1}$

AND

$\forall\ k,\ q+1 \leq k \leq m-1,\ c_k \leq e_{k+1} + c_{k+1}$

AND

$c_{q+1} \leq e_{q+2} + e_{q+3} + \cdots + e_m.$

then Sarkar's method finds the optimal partition Π_q.

Proof

Exactly the same as for the case of fork DAGs.

Theorem: Assume that the optimal partition of the fork DAG is Π_p, $1 \leq p \leq m$, and that Π_i, $0 \leq i \leq p-1$, is not an optimal partition.
If there exists an s, $1 \leq s \leq p$, such that merging edge (n_s, r) is not accepted (i.e. we have an increase in the CPL), then Sarkar's method does not find the optimal partition.

Proof

Exactly the same as for the case of fork DAGs.

6.2 Partitioning Complete Binary Trees

In this section, we assume that the program graph to be partitioned is a complete binary tree.
We also assume that all actors in the graph have the same weight, and all edges in the graph have the same weight.

Usefulness

Binary trees represent many useful problems, such as search, sort, finding the minium or maximum of a list, numerical algorithms constituted of binary operators only, and divide and conquer problem solving techniques.

Furthermore, it was shown that (see [15]) many efficient algorithms for several scheduling problems use binary trees (generally complete binary trees). Dekel and Sahni ([15]) used binary trees to design efficient parallel algorithms. More precisely, they used binary trees for parallel computations, and showed that the binary tree is an important and very useful program graph for parallel algorithms.

Definitions Related to Trees

Consider a directed graph constituted of a tree.
We assume that all leaf nodes are at the first level (top most, level 1) of the tree and the root node is at the last level (bottom level, level N where N is the number of levels in the tree) of the tree.
The tree is upside down and therefore the leaf nodes are at the top and the root node is at the bottom.
The direction of the arcs are from smaller to larger levels.

6.2.1 Properties

Property 1: G-Trees

Given a program graph g that is a complete binary tree, such that all actors in the graph have the same weight and all edges in the graph have the same weight. For any task graph T that corresponds to a partition of g, there is a task graph T' such that the CPL of T' is less or equal than the CPL of T, and T' has the following properties:
T' is a tree such that

- The number of input arcs of any node is a power of 2.
- All nodes (tasks) at the same level have the same number of actors (and as a consequence the same weight).
- The number of actors in any task (node in the task graph) is equal to the number of input edges of the node corresponding to the task minus 1 (clearly, this is not the case for nodes at level 1).
- The sum of the number of actors in all nodes (i.e. tasks corresponding to the nodes) in T' except the nodes at the top most level is $2^a - 1$, where 2^a is the number of nodes at the top most level of the tree.

T' is called a G-tree (G stands for Good).
Clearly all G-trees satisfy the following 2 properties:

- T' is a complete tree and all execution paths have the same length.
- All nodes at the top most level (level 1) represent tasks that have $2^{N-a} - 1$ actors, where N is the total number of levels in the tree and there are 2^a nodes at the top most level of the tree.

Property 2

This is a direct consequence of Property 1.
Given a program graph g that is a complete binary tree, such that all actors in the graph have the same weight and all edges in the graph have the same weight. The optimal task graph is a G-tree.

6.2.2 Optimal Partition

One way to figure out the optimal partition for complete binary trees with N levels is to find all possible task graphs which are G-trees and determine the one with the minimum CPL. That task graph is the optimal one (from Property 2).
Hence, we determine all possible G-trees with 1 level, all possible G-trees with 2 levels, ... , all possible G-trees with $N-1$ levels, and finally all possible G-trees with N levels.
Clearly the task graph consisting of a single node is the only G-tree with 1 level. Also the task graph corresponding to the trivial partition is the only G-tree with N levels.

Example

Assume that our program graph is a complete binary tree with 4 levels. Figure 6.12 shows all G-trees with 2 levels. Figure 6.13 shows all G-trees with 3 levels. The number next to each node is the number of actors in the task corresponding to the node.

CPL of G-trees

Assume that our program graph is a complete binary tree with N levels.
Consider all G-trees with m levels ($1 \leq m \leq N$).
For a path p, let $A(p)$ be the sum of the numbers of actors in all nodes in p. Clearly for any G-tree T, $A(p)$ is the same for any execution path p of T. Also any execution path of T is a critical path.
Let's define $A(T)$ to be $A(p)$, where p is any execution path of T.
For a G-tree T with m levels, the possible values for $A(T)$ are
$(2^{a_1}-1)+(2^{a_2}-1)+\cdots+(2^{a_m}-1)$, where $a_1, a_2, \ldots, a_m$ are positive integers (not zero) such that
$a_1+a_2+\cdots+a_m=N$.
$(2^{a_1}-1)$ is the number of actors in each node at level 1.
$(2^{a_2}-1)$ is the number of actors in each node at level 2.
....

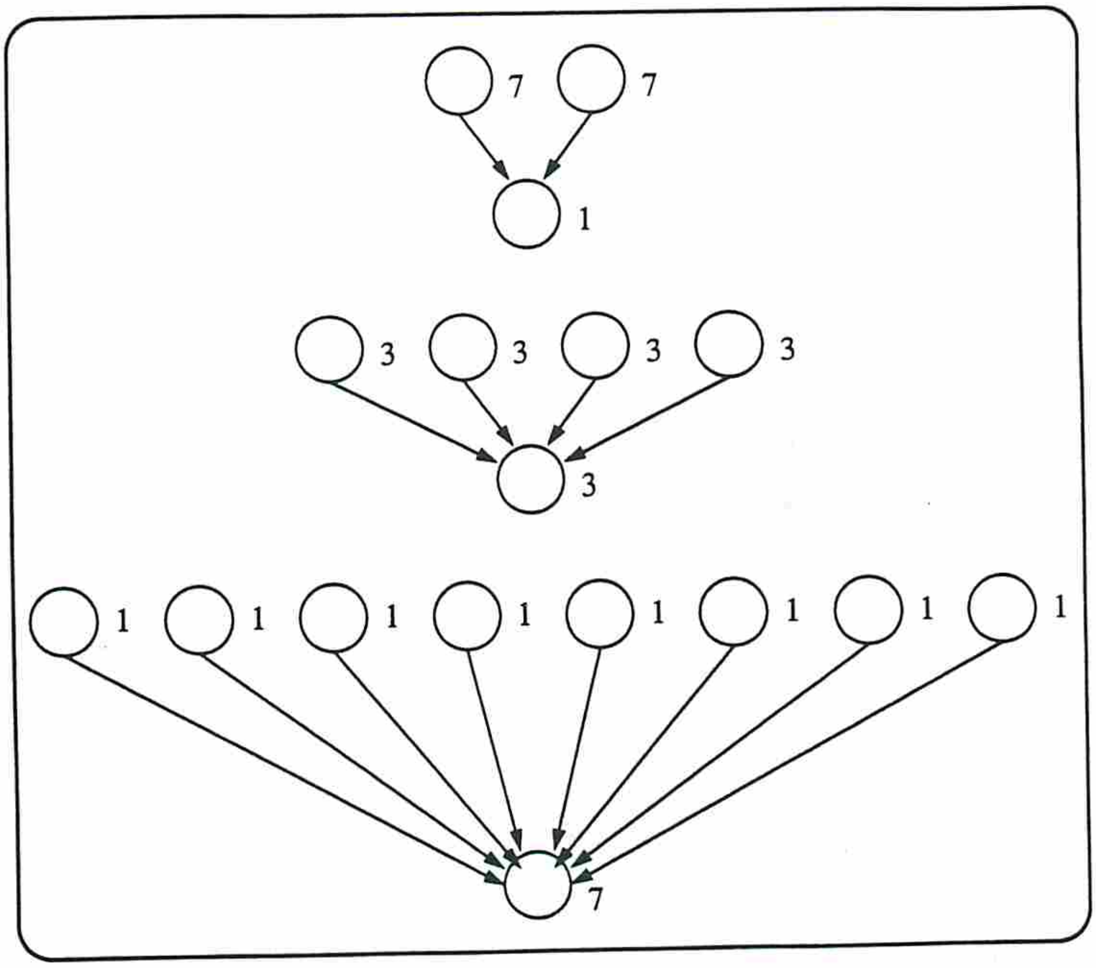


Figure 6.12: All G-trees with 2 levels

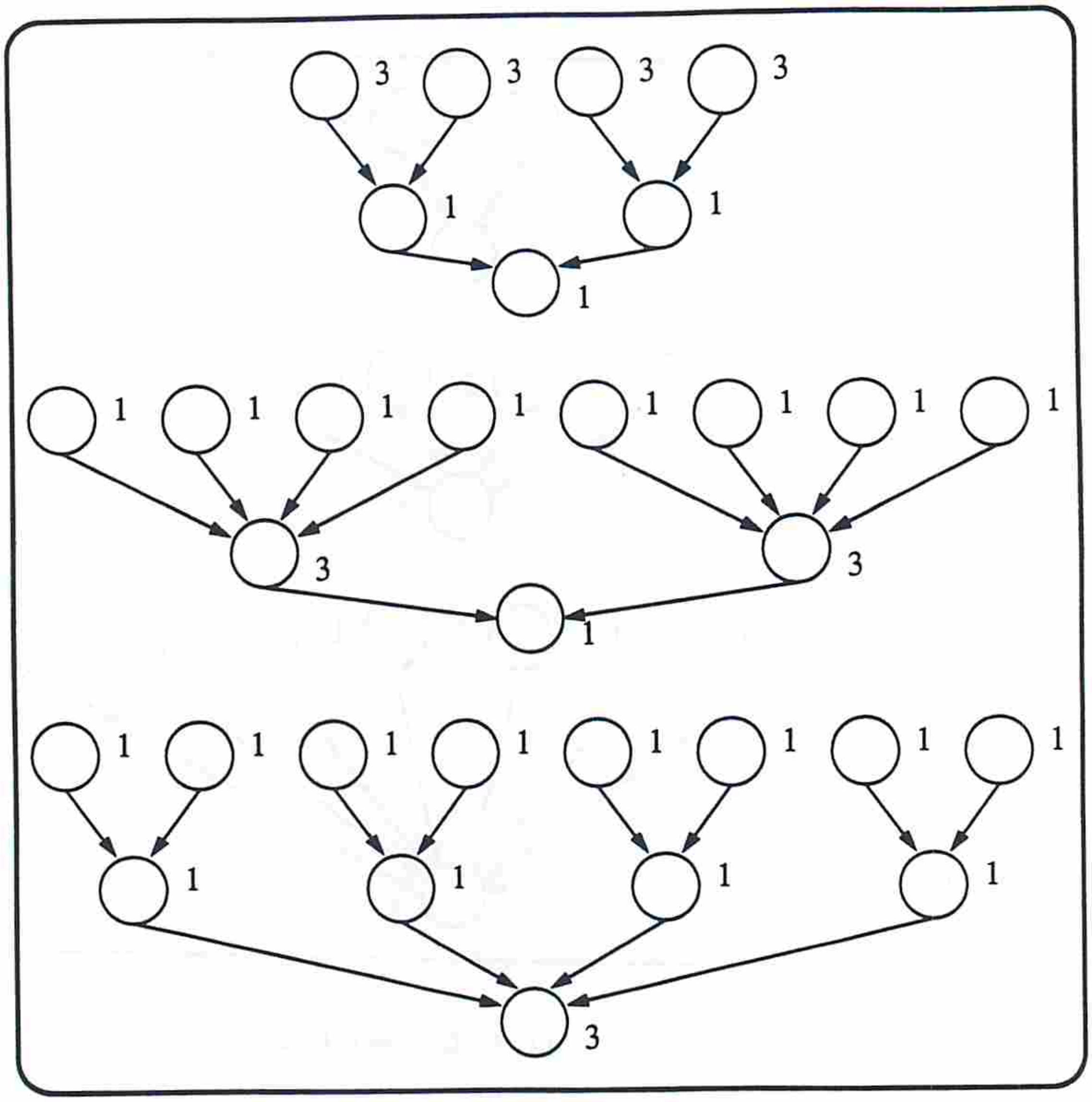


Figure 6.13: All G-trees with 3 levels

$(2^{a_m} - 1)$ is the number of actors in each node at level m.
Let e be the execution time of each actor in the original program graph.
Let c be the communication time of each edge in the graph[4].
Hence, the possible values for the CPL of T are
$CPL_m = [(2^{a_1} - 1) + (2^{a_2} - 1) + \cdots + (2^{a_m} - 1)] * e + (m - 1) * c$,
where $a_1, a_2, \ldots, a_m$ are positive integers (not zero) such that
$a_1 + a_2 + \cdots + a_m = N$.

Examples

1. Let $N = 4$ and $m = 3$.
   Let's find all possible values of (a_1, a_2, a_3).
   The solution of $a_1 + a_2 + a_3 = 4$ is: $\{(1,1,2),(1,2,1),(2,1,1)\}$.
   Hence there are 3 possible G-trees with 3 levels. These are shown in figure 6.13.

2. Let $N = 6$ and $m = 3$.
   In this case $a_1 + a_2 + a_3 = 6$.
   There are 10 possible values for (a_1, a_2, a_3).
   The solution is:
   $\{(1,1,4),(1,2,3),(1,3,2),(1,4,1),(2,1,3),(2,2,2),(2,3,1),(3,1,2),(3,2,1),(4,1,1)\}$.
   Hence there are 10 possible G-trees with 3 levels.

Minimal CPL

Assume that our program graph is a complete binary tree with N levels.
The G-tree with m levels which has the minimal CPL among all G-trees with m levels is one for which CPL_m is minimal. Hence, we find $a_1, a_2, \ldots, a_m$ such that
$A_m = 2^{a_1} + 2^{a_2} + \cdots + 2^{a_m}$ is minimal given that
$a_1, a_2, \ldots, a_m$ are positive integers (not zero) such that
$a_1 + a_2 + \cdots + a_m = N$.
A_m is minimal when the a_i's are chosen in the following manner:

[4]Note that the weight of the edges in any task graph is the same as the weight of the edges in the original program graph.

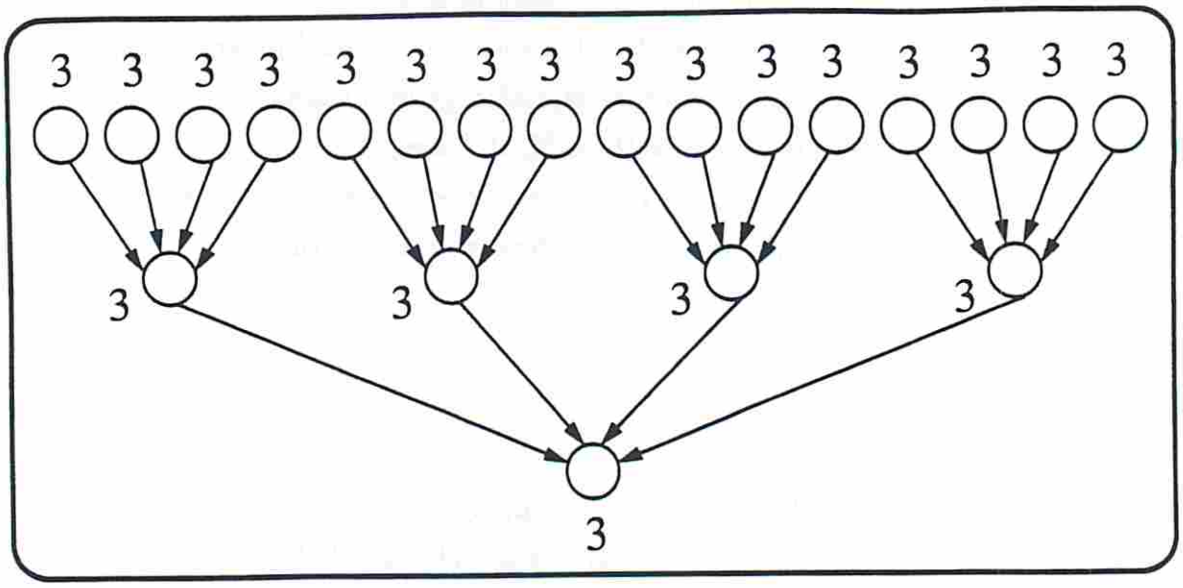


Figure 6.14: Optimal G-tree with 3 levels

We divide N as evenly as possible on $a_1, a_2, \ldots, a_m$. In other words, if N is a multiple of m, then each a_i will take the value N/m (using integer division). Otherwise, (N MOD m) a_i's will take the value $(N/m + 1)$ and the rest take the value N/m (using integer division). Hence when N is not a multiple of m, we have more than one solution for $(a_1, a_2, \ldots, a_m)$.

Examples

1. Let $N = 4$ and $m = 3$.
   The G-tree with 3 levels for which $(a_1, a_2, a_3) = (2, 1, 1)$ has minimal CPL among all G-trees with 3 levels.
   Also the G-tree with 3 levels for which $(a_1, a_2, a_3) = (1, 2, 1)$ has minimal CPL among all G-trees with 3 levels.

2. Let $N = 6$ and $m = 3$.
   The G-tree with 3 levels and $(a_1, a_2, a_3) = (2, 2, 2)$ has minimal CPL among all G-trees with 3 levels.

$(2^{a_1} - 1) = 3$ is the number of actors in each node at level 1.
$(2^{a_2} - 1) = 3$ is the number of actors in each node at level 2.
$(2^{a_3} - 1) = 3$ is the number of actors in each node at level 3.
Using property 1 for G-trees (first point), it is easy to construct this G-tree. This is shown in figure 6.14. The number next to each node is the number of actors in the task corresponding to the node.

Finding Optimal Partition

Given a complete binary tree with N levels as a program graph, one way to find the optimal partition is to determine:
1- Π_1: the G-tree with one level which has minimal CPL among all G-trees with one level.
2- Π_2: the G-tree with 2 levels which has minimal CPL among all G-trees with 2 levels.
....
m- Π_m: the G-tree with m levels which has minimal CPL among all G-trees with m levels.
....
N- Π_N: the G-tree with N levels which has minimal CPL among all G-trees with N levels.

The Π_i which has the smallest CPL among all Π_i's corresponds to the optimal partition.

Hence the CPL of the optimal partition can be expressed as
$CPL_{opt} =$

$$\begin{array}{lll} MIN \; \{ & (2^N - 1) * e, & \\ & MIN & \{[(2^{a_1} - 1) + (2^{a_2} - 1)] * e + c \; / \\ & & a_1 + a_2 = N\}, \\ & MIN & \{[(2^{a_1} - 1) + (2^{a_2} - 1) + \\ & & (2^{a_3} - 1)] * e + 2 * c \; / \\ & & a_1 + a_2 + a_3 = N\}, \\ & MIN & \{[(2^{a_1} - 1) + (2^{a_2} - 1) + \\ & & (2^{a_3} - 1) + (2^{a_4} - 1)] * e + 3 * c \; / \\ & & a_1 + a_2 + a_3 + a_4 = N\}, \\ & \ldots\ldots & \\ & MIN & \{[(2^{a_1} - 1) + (2^{a_2} - 1) + \\ & & \cdots + (2^{a_N} - 1)] * e + (N - 1) * c \; / \\ & & a_1 + a_2 + \cdots + a_N = N\} \\ \} & & \end{array}$$

Note that $\{[(2^{a_1}-1)+(2^{a_2}-1)+\cdots+(2^{a_N}-1)]*e+(N-1)*c \;/\; a_1+a_2+\cdots+a_N = N\} =$
$\{[(2^{a_1} - 1) + (2^{a_2} - 1) + \cdots + (2^{a_N} - 1)] * e + (N - 1) * c \;/\; a_1 = a_2 = \cdots = a_N = 1\}$
$=$
$\{N * e + (N - 1) * c\}$.

Remark: In general, there could be more than one optimal solution.

A Quicker Way to Find an Optimal Partition

In general, it is not necessary to find all Π_i's in order to find the optimal partition. There are two tricks to do that:

1. It is enough to find $\Pi_1, \Pi_2, \ldots, \Pi_m$, where m is such that any G-tree with $m + 1$ or more levels has a CPL greater or equal than l_{min}, and $l_{min} = MIN\{CPL(\Pi_i), 1 \leq i \leq m\}$, where $CPL(\Pi_i)$ is defined to be the CPL of

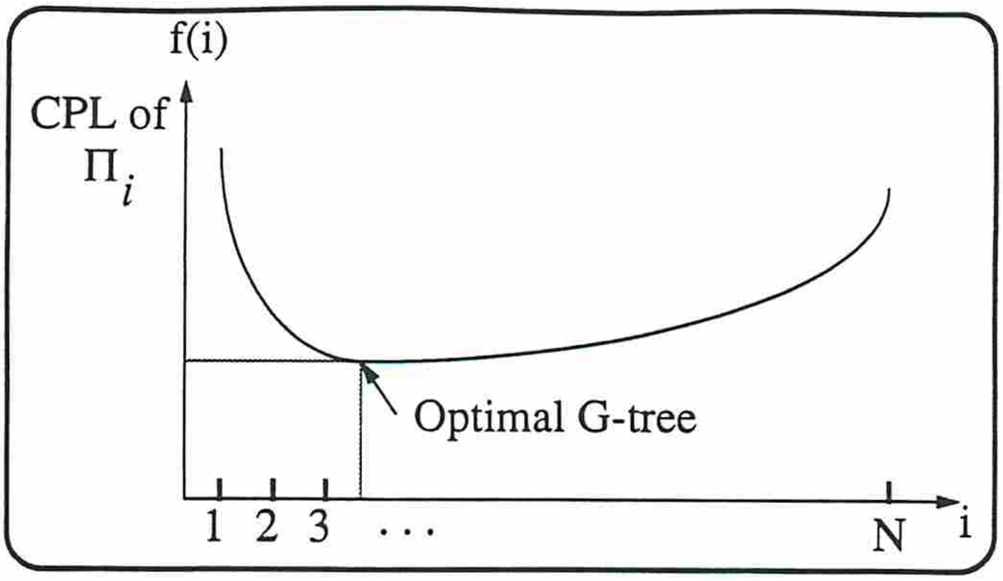


Figure 6.15: Plot of f(i)

partition Π_i.

Note that if N is large, this method will take too much time, and the next method should be used.

2. Let $f(i) := \text{CPL of } \Pi_i$.

   The general shape of the plot of $f(i)$ is shown in figure 6.15[5].

   Hence, all we need to do is to find Π_i such that $f(i-1) \geq f(i)$ and $f(i+1) > f(i)$. Π_i corresponds to the optimal partition[6].

   In general, by determining several points in the curve of the function f (i.e. we determine several Π_j's and their corresponding $f(j)$'s), we can determine i.

This is shown in the following examples.

[5]Note that it is possible for Π_1 to correspond to the optimal partition. For instance, if $(N, c, e) = (4, 10, 1)$ then Π_1 corresponds to the optimal partition.

[6]If Π_1 is the optimal G-tree, then $i = 1$ and $f(i-1)$ doesn't make any sense. If $f(2) > f(1)$ then we know that Π_1 is the optimal G-tree.

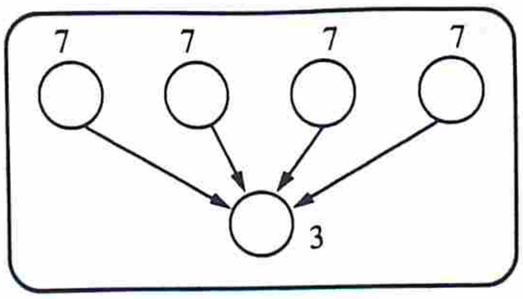


Figure 6.16: Optimal task graph

Example 1

Assume that our program graph is a complete binary tree with 5 levels.
The total number of nodes in the graph is $2^5 - 1 = 31$.
Assume that the execution time of the actors is 1 and the communication cost of edges is 5.

1. The G-tree with one level is constituted of 1 node. The corresponding task of this node contains 31 actors. Hence the CPL is 31.

2. G-tree with 2 levels that has minimum CPL:
   $a_1 = 5/2 + 1 = 3$.
   $a_2 = 5/2 = 2$.
   The corresponding CPL is $[(2^3 - 1) + (2^2 - 1)] * 1 + (2 - 1) * 5 = 15$.

3. G-tree with 3 levels that has minimum CPL:
   $a_1 = 5/3 + 1 = 2$.
   $a_2 = 5/3 + 1 = 2$.
   $a_3 = 5/3 = 1$.
   The corresponding CPL is $[(2^2 - 1) + (2^2 - 1) + (2^1 - 1)] * 1 + (3 - 1) * 5 = 17$.

4. G-tree with 4 levels that has minimum CPL:
   $a_1 = 5/4 + 1 = 2$.
   $a_2 = 5/4 = 1$.

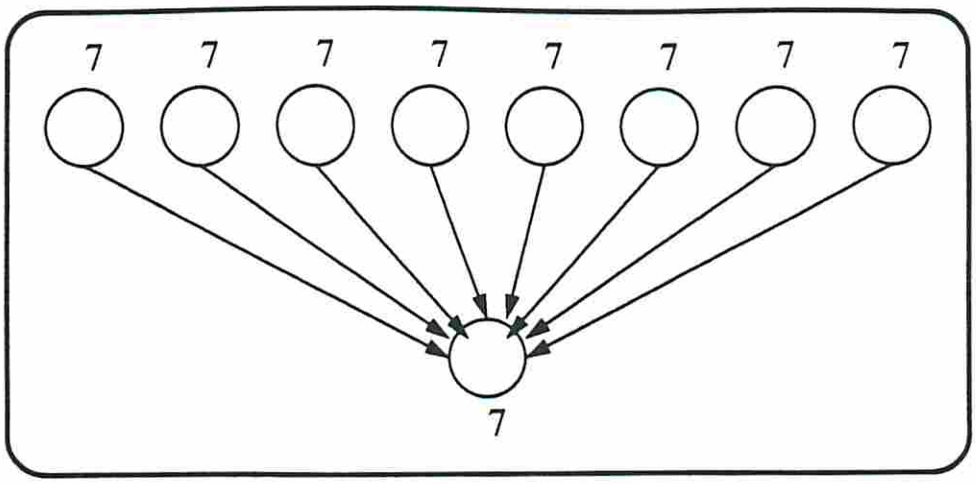


Figure 6.17: Optimal task graph

$a_3 = 5/4 = 1$.
$a_4 = 5/4 = 1$.
The corresponding CPL is $[(2^2-1)+(2^1-1)+(2^1-1)+(2^1-1)]*1+(4-1)*5$ $= 21$.

5. The G-tree with 5 levels is the original program graph.
Its corresponding CPL is $(1+1+1+1+1)*1+(5-1)*5 = 25$.

Hence, the G-tree with 2 levels and $(a_1, a_2) = (3, 2)$ is an optimal task graph (i.e. the partition it represents is optimal). This G-tree is shown in figure 6.16. The number next to each node is the number of actors in the task corresponding to the node.

Remark: It was not necessary to look at G-trees with levels 4 and 5. We could have stopped when we found the best G-tree with 1,2 or 3 levels. This is true for the following reason: knowing that the best G-tree (i.e. the one that has minimal CPL) with 1, 2 or 3 levels has a CPL of 15, and that any G-tree with 4 or more levels has a CPL greater or equal than $(4*1+3*5) = 19$, we can conclude that the best G-tree found for 1, 2 or 3 levels is the optimal task graph.

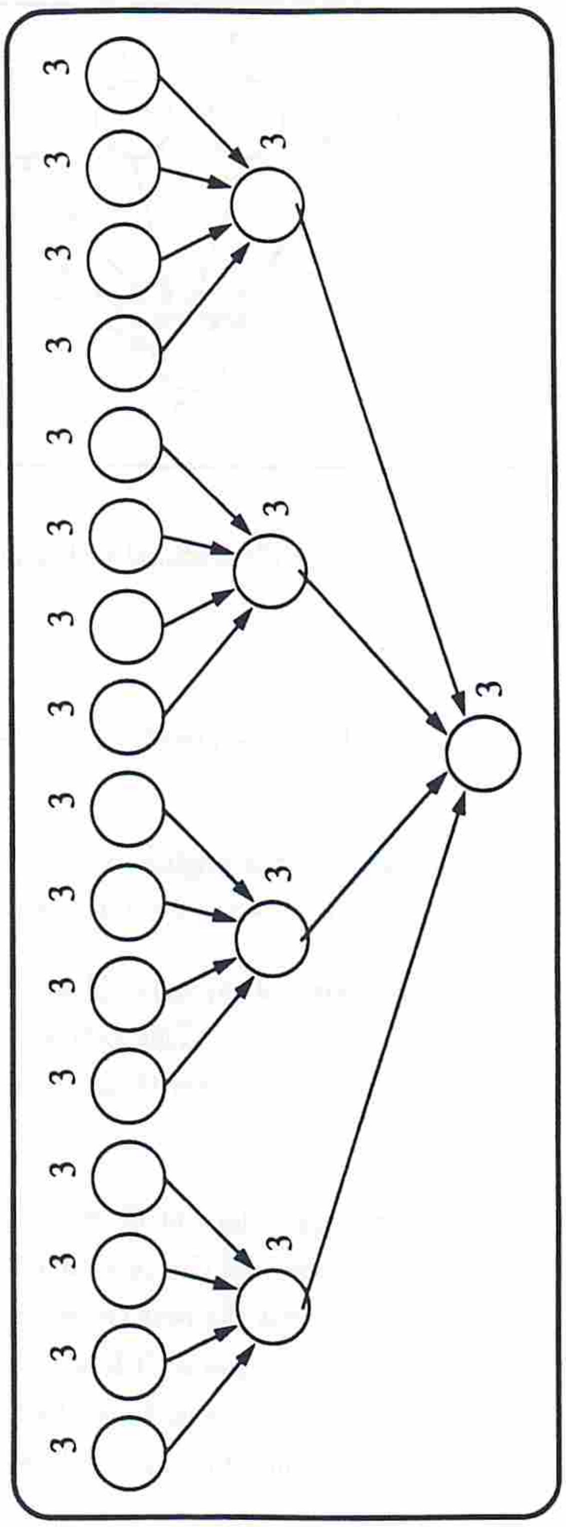


Figure 6.18: Optimal task graph

Example 2

Assume that our program graph is a complete binary tree with 6 levels.
The total number of nodes in the graph is $2^6 - 1 = 63$.
Assume that the execution time of the actors is 1 and the communication cost of edges is 5.

1. The G-tree with one level is constituted of 1 node. The corresponding task of this node contains 63 actors. Hence the CPL is 63.

2. G-tree with 2 levels that has minimum CPL:
   $a_1 = 6/2 = 3$.
   $a_2 = 6/2 = 3$.
   The corresponding CPL is $[(2^3 - 1) + (2^3 - 1)] * 1 + (2 - 1) * 5 = 19$.

3. G-tree with 3 levels that has minimum CPL:
   $a_1 = 6/3 = 2$.
   $a_2 = 6/3 = 2$.
   $a_3 = 6/3 = 2$.
   The corresponding CPL is $[(2^2 - 1) + (2^2 - 1) + (2^2 - 1)] * 1 + (3 - 1) * 5 = 19$.

4. G-tree with 4 levels that has minimum CPL:
   $a_1 = 6/4 + 1 = 2$.
   $a_2 = 6/4 + 1 = 2$.
   $a_3 = 6/4 = 1$.
   $a_4 = 6/4 = 1$.
   The corresponding CPL is $[(2^2-1)+(2^2-1)+(2^1-1)+(2^1-1)]*1+(4-1)*5 = 23$.

5. Any G-tree with 5 or more levels:
   Its corresponding CPL is greater or equal than $(1+1+1+1+1)*1+(5-1)*5$.
   Thus CPL ≥ 25.

Hence, the G-tree with 2 levels and $(a_1, a_2) = (3, 3)$ is an optimal task graph. This G-tree is shown in figure 6.17.

Also, the G-tree with 3 levels and $(a_1, a_2, a_3) = (2, 2, 2)$ is an optimal task graph. This G-tree is shown in figure 6.18.

Example 3

Assume that our program graph is a complete binary tree with 5 levels.
The total number of nodes in the graph is $2^5 - 1 = 31$.
Assume that the execution time of the actors is 1 and the communication cost of edges is 10.

1. The G-tree with one level is constituted of 1 node. The corresponding task of this node contains 31 actors. Hence the CPL is 31.

2. G-tree with 2 levels that has minimum CPL:
   $a_1 = 3$.
   $a_2 = 2$.
   The corresponding CPL is $[(2^3 - 1) + (2^2 - 1)] * 1 + (2 - 1) * 10 = 20$.

3. G-tree with 3 or more levels:
   The corresponding CPL is greater or equal than $(1 + 1 + 1) * 1 + (3 - 1) * 10$.
   Hence CPL ≥ 23.

Hence, the G-tree with 2 levels and $(a_1, a_2) = (3, 2)$ is an optimal task graph (i.e. the partition it represents is optimal).
This G-tree is shown in figure 6.16.

Example 4

Assume that our program graph is a complete binary tree with 6 levels.
The total number of nodes in the graph is $2^6 - 1 = 63$.
Assume that the execution time of the actors is 1 and the communication cost of edges is 10.

1. The G-tree with one level is constituted of 1 node. The corresponding task of this node contains 63 actors. Hence the CPL is 63.

2. G-tree with 2 levels that has minimum CPL:

   $a_1 = 3$.

   $a_2 = 3$.

   The corresponding CPL is $[(2^3 - 1) + (2^3 - 1)] * 1 + (2 - 1) * 10 = 24$.

3. G-tree with 3 levels that has minimum CPL:

   $a_1 = 2$.

   $a_2 = 2$.

   $a_3 = 2$.

   The corresponding CPL is $[(2^2 - 1) + (2^2 - 1) + (2^2 - 1)] * 1 + (3 - 1) * 10$ $= 29$.

4. G-tree with 4 or more levels:

   The corresponding CPL is greater or equal than $(1+1+1+1)*1+(4-1)*10$.

   Hence CPL ≥ 34.

Hence, the G-tree with 2 levels and $(a_1, a_2) = (3, 3)$ is an optimal task graph. This G-tree is shown in figure 6.17.

Example 5

Assume that our program graph is a complete binary tree with 100 levels.
The total number of nodes in the graph is $2^{100} - 1$.
Assume that the execution time of the actors is 1 and the communication cost of edges is 10.

- Π_{10}: $a_1 = a_2 = \cdots = a_{10} = 10$.
  $f(10) = 10320$.

- Π_{20}: $a_1 = a_2 = \cdots = a_{20} = 5$.
  $f(20) = 810$.

- Π_{50}: $a_1 = a_2 = \cdots = a_{50} = 2$.
  $f(50) = 640$.

- Π_{25}: $a_1 = a_2 = \cdots = a_{25} = 4$.
  $f(25) = 615$.

- Π_{30}: $a_1 = a_2 = \cdots = a_{10} = 4.$
  $a_{11} = a_{12} = \cdots = a_{30} = 3.$
  $f(30) = 580.$
- Π_{40}: $a_1 = a_2 = \cdots = a_{20} = 3.$
  $a_{21} = a_{22} = \cdots = a_{40} = 2.$
  $f(40) = 590.$
- Π_{35}: $a_1 = a_2 = \cdots = a_{30} = 3.$
  $a_{31} = a_{32} = \cdots = a_{35} = 2.$
  $f(35) = 565.$
- Π_{38}: $a_1 = a_2 = \cdots = a_{24} = 3.$
  $a_{25} = a_{26} = \cdots = a_{38} = 2.$
  $f(38) = 580.$
- Π_{36}: $a_1 = a_2 = \cdots = a_{28} = 3.$
  $a_{29} = a_{30} = \cdots = a_{36} = 2.$
  $f(36) = 570.$
- Π_{34}: $a_1 = a_2 = \cdots = a_{32} = 3.$
  $a_{33} = a_{34} = 2.$
  $f(34) = 560.$
- Π_{33}: $a_2 = a_3 = \cdots = a_{33} = 3.$
  $a_1 = 4.$
  $f(33) = 559.$
- Π_{32}: $a_1 = a_2 = a_3 = a_4 = 4.$
  $a_5 = a_6 = \cdots = a_{32} = 3.$
  $f(32) = 566.$

Hence, Π_{33} is the optimal G-tree and the optimal CPL is 559.

Example 6

Assume that our program graph is a complete binary tree with 50 levels.
The total number of nodes in the graph is $2^{50} - 1$.
Assume that the execution time of the actors is 1 and the communication cost of edges is 10.

- Π_{25}: $a_1 = a_2 = \cdots = a_{25} = 2$.
  $f(25) = 315$.

- Π_{20}: $a_1 = a_2 = \cdots = a_{10} = 3$.
  $a_{11} = a_{12} = \cdots = a_{20} = 2$.
  $f(20) = 290$.

- Π_{15}: $a_1 = a_2 = \cdots = a_5 = 4$.
  $a_6 = a_7 = \cdots = a_{15} = 3$.
  $f(15) = 285$.

- Π_{14}: $a_1 = a_2 = \cdots = a_8 = 4$.
  $a_9 = a_{10} = \cdots = a_{14} = 3$.
  $f(14) = 292$.

- Π_{16}: $a_1 = a_2 = 4$.
  $a_3 = a_4 = \cdots = a_{16} = 3$.
  $f(16) = 278$.

- Π_{17}: $a_1 = a_2 = \cdots = a_{16} = 3$.
  $a_{17} = 2$.
  $f(17) = 275$.

- Π_{18}: $a_1 = a_2 = \cdots = a_{14} = 3$.
  $a_{15} = a_{16} = \cdots = a_{18} = 2$.
  $f(18) = 280$.

Hence, Π_{17} is the optimal G-tree and the optimal CPL is 275.

6.2.3 Using Heuristic 1

Example 1

Assume that our program graph is a complete binary tree with 3 levels.
Assume that the execution time of the actors is 1 and the communication cost of edges is 2.
Figure x31 shows the merging steps using Heuristic 1. The number next to each node is the number of actors in the task corresponding to the node. The edge that has an "x" mark next to it is the edge chosen for merger. From the figure, we see that the task graph that has a CPL of 6 corresponds to the the best partition using Heuristic 1.

Example 2

Assume that our program graph is a complete binary tree with 5 levels.
Assume that the execution time of the actors is 1 and the communication cost of edges is 5.
Figure x32 shows the merging steps using Heuristic 1. From the figure, we see that the task graph that has a CPL of 19 corresponds to the the best partition using Heuristic 1.

Example 3

Assume that our program graph is a complete binary tree with 6 levels.
Assume that the execution time of the actors is 1 and the communication cost of edges is 10.
Figure x33 shows the merging steps using Heuristic 1. From the figure, we see that the task graph that has a CPL of 37 corresponds to the the best partition using Heuristic 1.

Example 4

Assume that our program graph is a complete binary tree with 8 levels.
Assume that the execution time of the actors is 1 and the communication cost of

edges is 15.
Figure x34 shows the merging steps using Heuristic 1. From the figure, we see that the task graph that has a CPL of 79 corresponds to the the best partition using Heuristic 1. In the last task graph shown in the figure there are two "x" marks to indicate that both execution paths to which the marks belong are critical paths, and therefore either of the edges marked could be chosen for merger (clearly these are not the only edges that could be chosen).

Best Partition Using Heuristic 1

Assume that the execution time of actors is e and the communication cost of edges is c.

We find the smallest integer x such that $x \geq 1$ and $(2^x - 1) * e \geq c$.
Hence, we find the smallest integer x such that $x \geq 1$ and $2^x \geq \frac{c}{e} + 1$.
The best partition using Heuristic 1 is the one for which the task graph is a complete binary tree with $[N - (x - 1)]$ levels such that all top most nodes have $(2^x - 1)$ actors and all other nodes have 1 actor.
Hence, the corresponding CPL is

$$\begin{aligned} CPL &= [(2^x - 1) + ((N - (x - 1)) - 1)] * e + [(N - (x - 1)) - 1] * c \\ &= (2^x - 1 + N - x) * e + (N - x) * c \end{aligned}$$

Example 1

Assume that our program graph is a complete binary tree with 5 levels.
Assume that the execution time of the actors is 1 and the communication cost of edges is 10.
Find the smallest integer x such that $x \geq 1$ and $2^x \geq 11$.
$x = 4$.
Hence, the best partition using Heuristic 1 is the one for which the task graph is a complete binary tree with $5 - 3 = 2$ levels such that all top most nodes have $2^4 - 1 = 15$ actors and all other nodes have 1 actor.

The corresponding CPL is
$CPL = (2^4 - 1 + 5 - 4) * 1 + (5 - 4) * 10 = 26.$

Example 2

Assume that our program graph is a complete binary tree with 6 levels.
Assume that the execution time of the actors is 1 and the communication cost of edges is 5.
Find the smallest integer x such that $x \geq 1$ and $2^x \geq 6$.
$x = 3$.
Hence, the best partition using Heuristic 1 is the one for which the task graph is a complete binary tree with $6 - 2 = 4$ levels such that all top most nodes have $2^3 - 1 = 7$ actors and all other nodes have 1 actor.
The corresponding CPL is
$CPL = (2^3 - 1 + 6 - 3) * 1 + (6 - 3) * 5 = 25.$

Example 3

Assume that our program graph is a complete binary tree with 4 levels.
Assume that the execution time of the actors is 1 and the communication cost of edges is 10.
Find the smallest integer x such that $x \geq 1$ and $2^x \geq 11$.
$x = 4$.
Hence, the best partition using Heuristic 1 is the one for which the task graph is a complete binary tree with $4 - 3 = 1$ level such that all top most nodes have $2^4 - 1 = 15$ actors and all other nodes have 1 actor.
2, this is the task graph constituted of 1 node that has all 15 actors.
The corresponding CPL is
$CPL = (2^4 - 1 + 4 - 4) * 1 + (4 - 4) * 5 = 15.$

Example 4

Assume that our program graph is a complete binary tree with 8 levels.
Assume that the execution time of the actors is 1 and the communication cost of

edges is 5.

Find the smallest integer x such that $x \geq 1$ and $2^x \geq 6$.

$x = 3$.

Hence, the best partition using Heuristic 1 is the one for which the task graph is a complete binary tree with $8 - 2 = 6$ levels such that all top most nodes have $2^3 - 1 = 7$ actors and all other nodes have 1 actor.

The corresponding CPL is

$CPL = (2^3 - 1 + 8 - 3) * 1 + (8 - 3) * 5 = 37$.

Example 5

Assume that our program graph is a complete binary tree with 100 levels.

Assume that the execution time of the actors is 1 and the communication cost of edges is 10.

Find the smallest integer x such that $x \geq 1$ and $2^x \geq 11$.

$x = 4$.

Hence, the best partition using Heuristic 1 is the one for which the task graph is a complete binary tree with $100 - 3 = 97$ levels such that all top most nodes have $2^4 - 1 = 15$ actors and all other nodes have 1 actor.

The corresponding CPL is

$CPL = (2^4 - 1 + 100 - 4) * 1 + (100 - 4) * 10 = 1071$.

Example 6

Assume that our program graph is a complete binary tree with 50 levels.

Assume that the execution time of the actors is 1 and the communication cost of edges is 10.

Find the smallest integer x such that $x \geq 1$ and $2^x \geq 11$.

$x = 4$.

Hence, the best partition using Heuristic 1 is the one for which the task graph is a complete binary tree with $50 - 3 = 47$ levels such that all top most nodes have $2^4 - 1 = 15$ actors and all other nodes have 1 actor.

The corresponding CPL is

$CPL = (2^4 - 1 + 50 - 4) * 1 + (50 - 4) * 10 = 521.$

6.2.4 Performance of Heuristic 1

Assume that we have a complete binary tree with N levels.
Let the communication cost of edges be $c = 10$ and the execution cost of actors be $e = 1$.

We determine the performance of Heuristic 1 by comparing the partition obtained using this heuristic with the optimal partition for various values of N.
We assume that the performance of a partitioning algorithm is the inverse of the CPL of the task graph corresponding to the partition obtained using the algorithm.
Let l be the CPL of the task graph corresponding to the partition obtained using Heuristic 1 and l_{opt} be the CPL of the task graph corresponding to the optimal partition.
Let p be the percentage of the performance of Heuristic 1 relative to the optimal partitioning algorithm.
$\frac{1}{l_{opt}} * \frac{p}{100} = \frac{1}{l} \Rightarrow p = \frac{l_{opt}}{l} * 100.$

- $N = 4.$
  Optimal: Π_1.
  $l_{opt} = 15.$
  $l = 15.$
  $p = 100$

- $N = 5.$
  Optimal: Π_2.
  $l_{opt} = 20.$
  $l = 26.$
  $p = 76.9$

- $N = 6.$
  Optimal: Π_2.

$l_{opt} = 24.$
$l = 37.$
$p = 64.9$

- $N = 8.$
  Optimal: Π_3.
  $l_{opt} = 37.$
  $l = 59.$
  $p = 62.7$

- $N = 10.$
  Optimal: Π_3.
  $l_{opt} = 49.$
  $l = 81.$
  $p = 60.5$

- $N = 20.$
  Optimal: Π_7.
  $l_{opt} = 105.$
  $l = 191.$
  $p = 55$

- $N = 30.$
  Optimal: Π_{10}.
  $l_{opt} = 160.$
  $l = 301.$
  $p = 53.2$

- $N = 40.$
  Optimal: Π_{13}.
  $l_{opt} = 219.$
  $l = 411.$
  $p = 53.3$

- $N = 50.$
  Optimal: Π_{17}.

N	4	5	6	8	10	20	30	40	50	100	150	200
l_{opt}	15	20	24	37	49	105	160	219	275	559	840	1125
l	15	26	37	59	81	191	301	411	521	1071	1621	2171
p	100	76.9	64.9	62.7	60.5	55.0	53.2	53.3	52.8	52.2	51.8	51.8

Table 6.1: Performance of Heuristic 1 for Complete Binary Trees

$l_{opt} = 275.$

$l = 521.$

$p = 52.8$

- $N = 100.$

  Optimal: Π_{33}.

  $l_{opt} = 559.$

  $l = 1071.$

  $p = 52.2$

- $N = 150.$

  Optimal: Π_{50}.

  $l_{opt} = 840.$

  $l = 1621.$

  $p = 51.81986 \approx 51.82$

- $N = 200.$

  Optimal: Π_{67}.

  $l_{opt} = 1125.$

  $l = 2171.$

  $p = 51.81943 \approx 51.82$

Table 6.1 summarizes the above results.

Figure 6.19 shows the plot of p as a function of N.
We see from the curve that as N exceeds 25, p starts to decrease very slowly. When N reaches 50, the decrease in p becomes almost negligible.
We can safely conclude that the performance of Heuristic 1 is above 50% of the performance of the optimal partitioning algorithm, for any value of N.

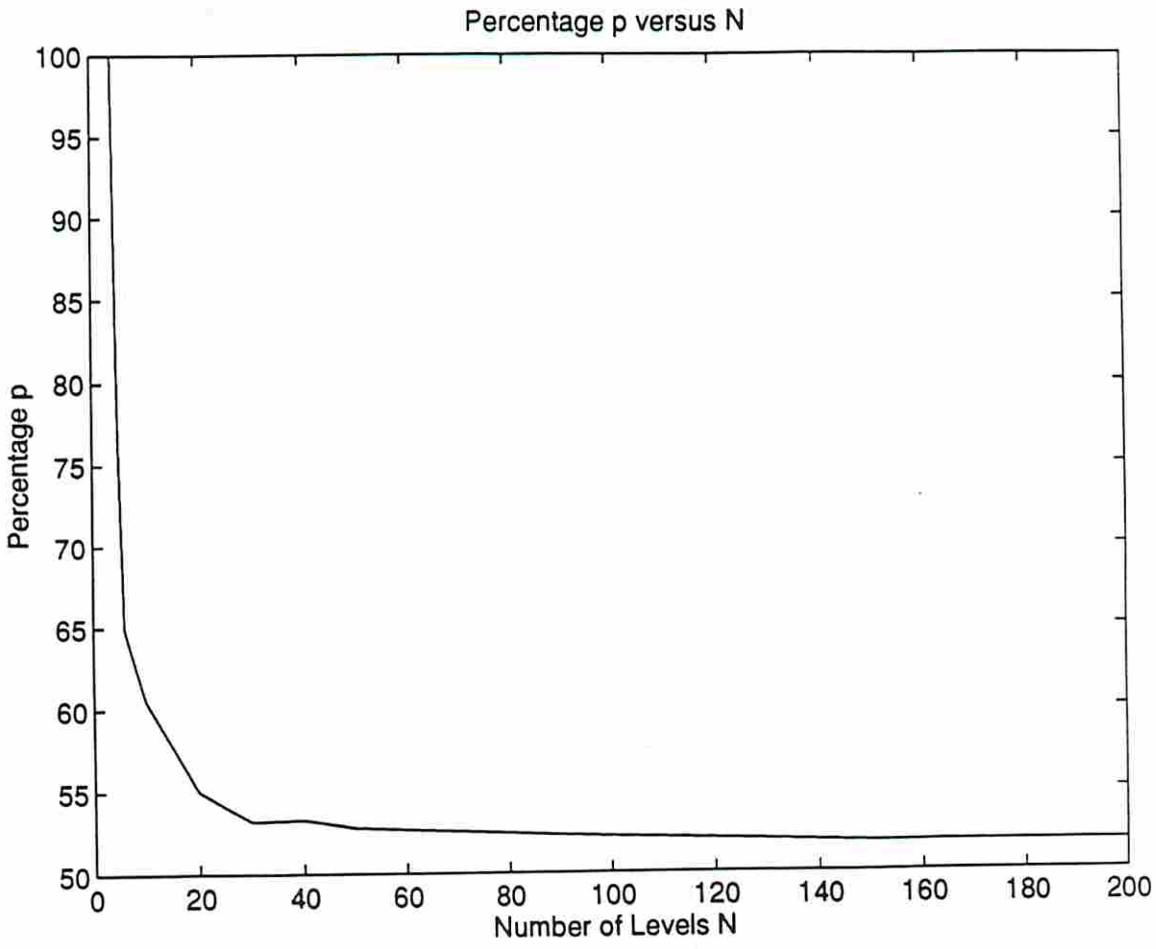


Figure 6.19: Performance of Heuristic 1

6.2.5 Using Sarkar's Partitioning Method

Since all edges in the graph have the same weight, Sarkar's method chooses edges to be merged randomly. Clearly, this could result in very poor performance.

6.3 Merger that Results into Higher PARTIME

In this section, we show why we have to keep doing the merger until the coarsest partition is obtained. We will see that we accept the merger of the edge chosen, even if it results into a higher CPL (PARTIME).

As was seen earlier, the parallel execution time of the program is PARTIME $= T_c + T_o$, where
$T_c :=$ Computation Time Component, and
$T_o :=$ Overhead Component (communication overhead only, no scheduling overhead).
There is a trade-off between computation component and overhead component. The more parallelism we exploit, the smaller T_c and the larger T_o will be, and vice versa. In general, merging tasks results into an increase in T_c (loss in parallelism and more sequentialization) and a decrease in T_o (reduction in communication overhead).
The CPL of the task graph is $CPL = \sum_{n \in P_{crit}} comp(n) + \sum_{e \in P_{crit}} comm(e)$. Clearly,
$T_c = \sum_{n \in P_{crit}} comp(n)$. and
$T_o = \sum_{e \in P_{crit}} comm(e)$.

The partitioning algorithm consists of a loop in which each iteration consists of a merging step. Assume that there are N iterations in the algorithm.
Let Π_i be the partition obtained after iteration i is done[7]. Π_N is the coarsest partition (i.e. the singleton partition).
Let $l(i)$ be the CPL of the task graph at the end of iteration i, $1 \leq i \leq N$, of our partitioning algorithm (using Heuristic 1). $l(0)$ is the CPL of the initial task

[7]Iteration 0 is not defined. Π_0 is defined to be the initial partition, before any iteration is performed.

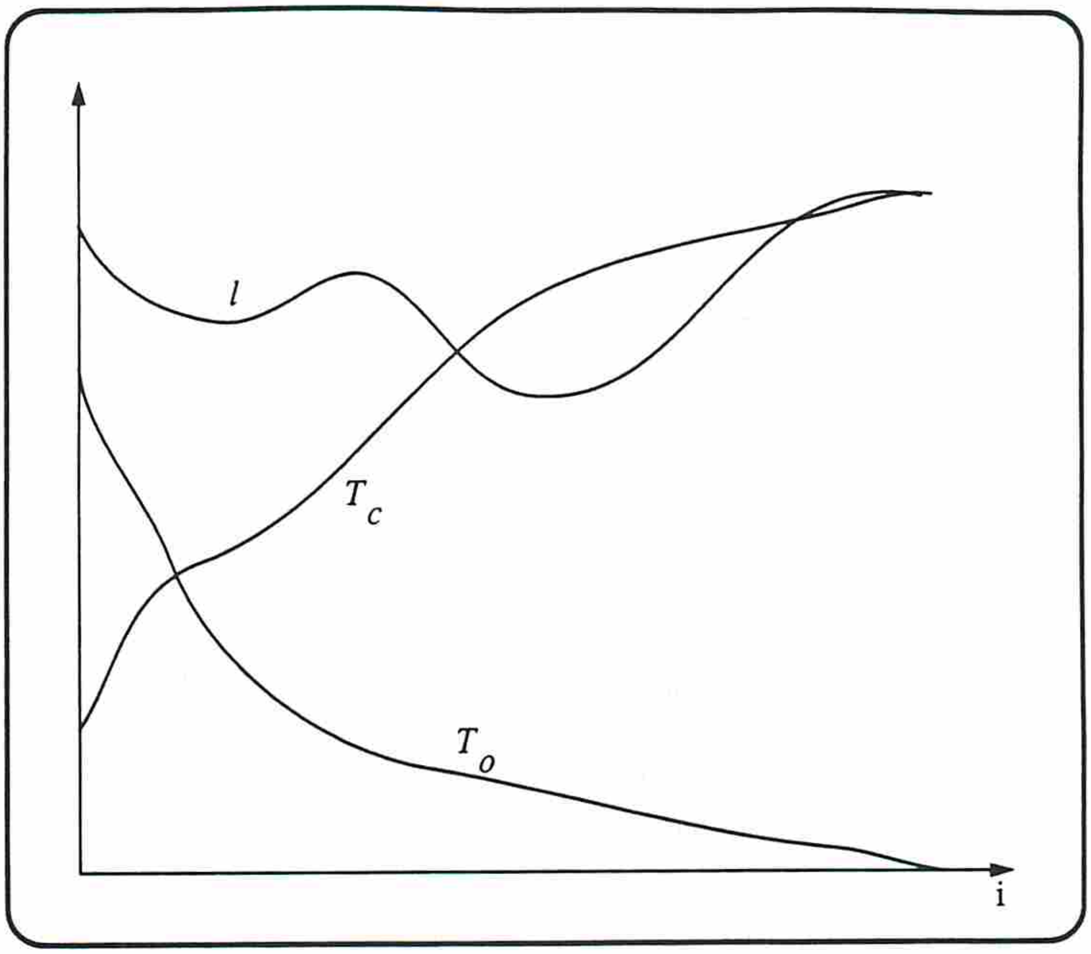


Figure 6.20: CPL as a function of the iteration number

graph.
$l(i) = T_c(i) + T_o(i)$, where $T_c(i)$ and $T_o(i)$ are the computation component and overhead component at the end of iteration i respectively.

Intuitively, it is easy to see that during the execution of the algorithm, T_c starts to increase and T_o starts to decrease. The net result (i.e. the sum of the 2 components) is shown in figure 6.20. From the curve shown in figure 6.20, we can see that PARTIME can increase momentarily, then decrease. Therefore, if during an iteration of the algorithm the merger causes an increase in PARTIME, we should still accept this merger, and continue with the next merging iterations.

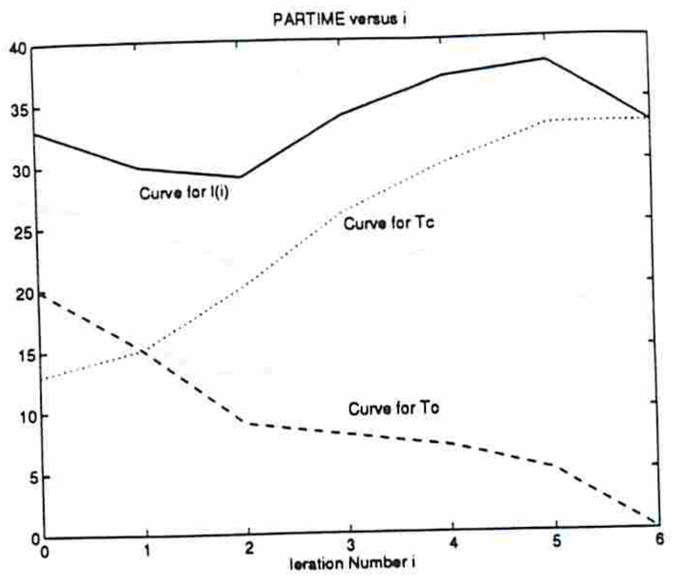


Figure 6.21: PARTIME Plot for Example 1

Otherwise we can get caught at a *local minimum*. It is for this reason that we keep merging tasks until the coarsest partition (singleton partition) is reached.

Examples

We consider the case of the fork DAG shown in figure 6.1.
$T_c(i) = r + n_1 + n_2 + \cdots + n_i + n_{i+1}$, $0 \leq i \leq m - 1$.
$T_c(m) = r + n_1 + n_2 + \cdots + n_m$.
$T_o(i) = c_{i+1}$, $0 \leq i \leq m - 1$.
$T_o(m) = 0$.

1. Consider the fork DAG shown in figure 6.9 (a). Here $N = 6$. The values of T_c, T_o and $l(i)$ for each iteration of the algorithm are shown in table 6.2.

   The plots for PARTIME, T_c and T_o are shown in figure 6.21.

2. Consider the fork DAG shown in figure 6.9 (b). Here $N = 6$. The values of T_c, T_o and $l(i)$ for each iteration of the algorithm are shown in table 6.3.

   The plots for PARTIME, T_c and T_o are shown in figure 6.22.

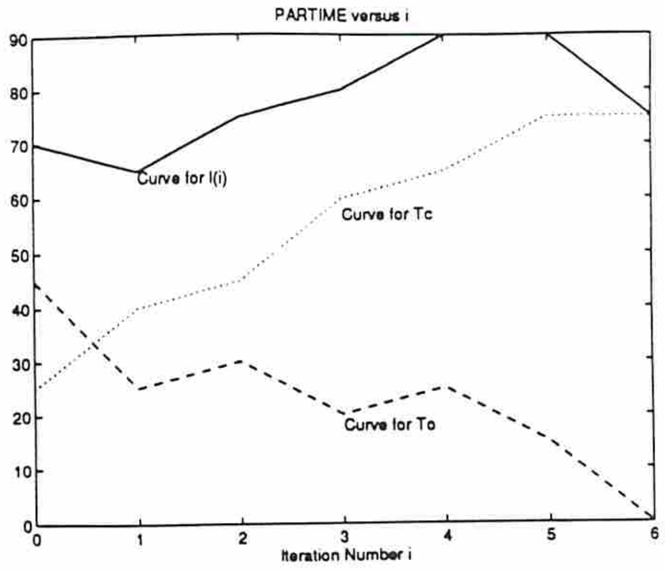


Figure 6.22: PARTIME Plot for Example 2

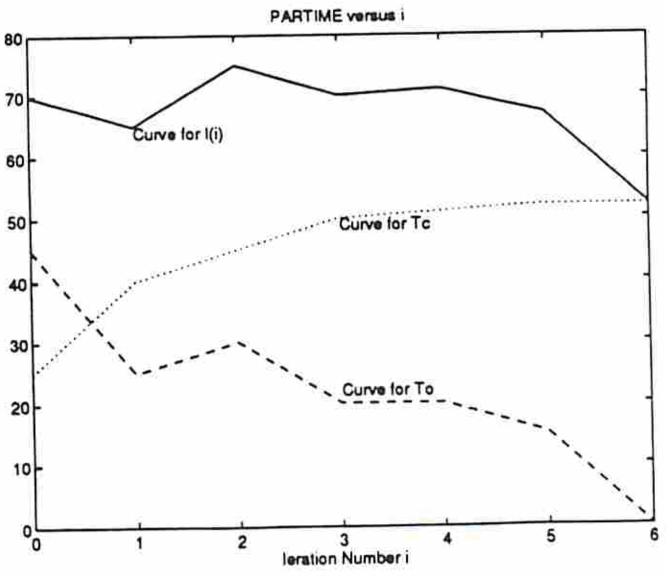


Figure 6.23: PARTIME Plot for Example 3

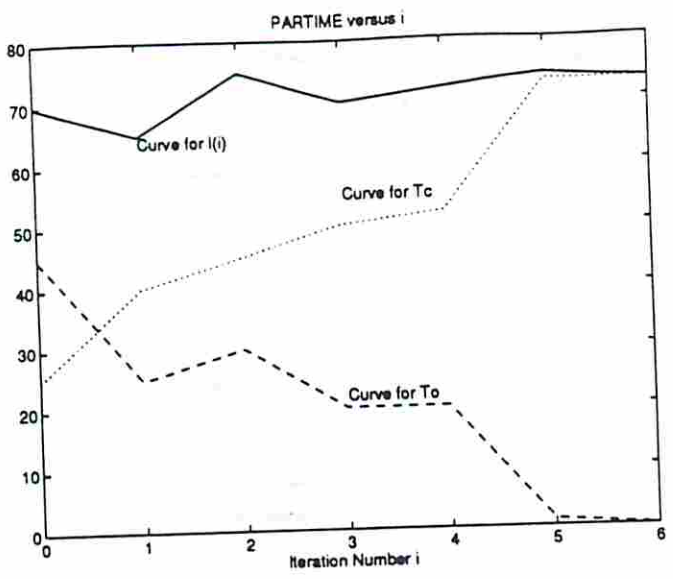


Figure 6.24: PARTIME Plot for Example 4

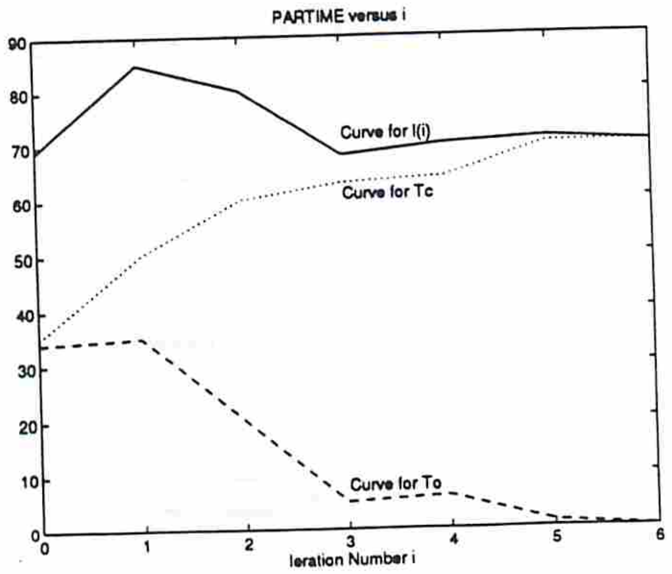


Figure 6.25: PARTIME Plot for Example 5

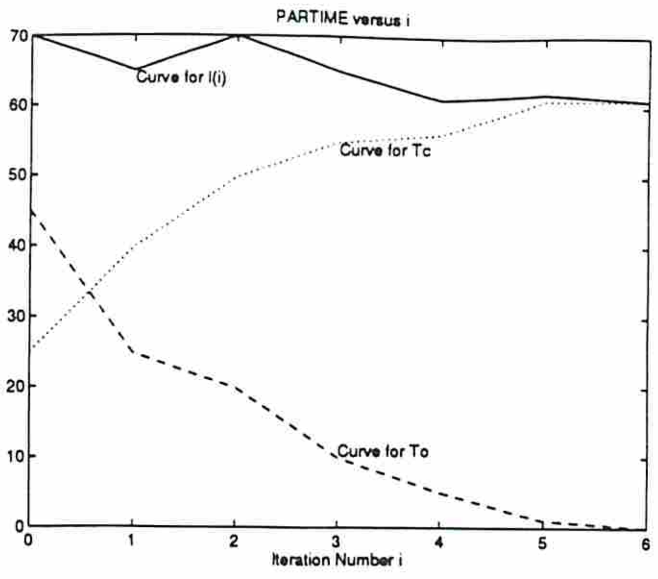


Figure 6.26: PARTIME Plot for Example 6

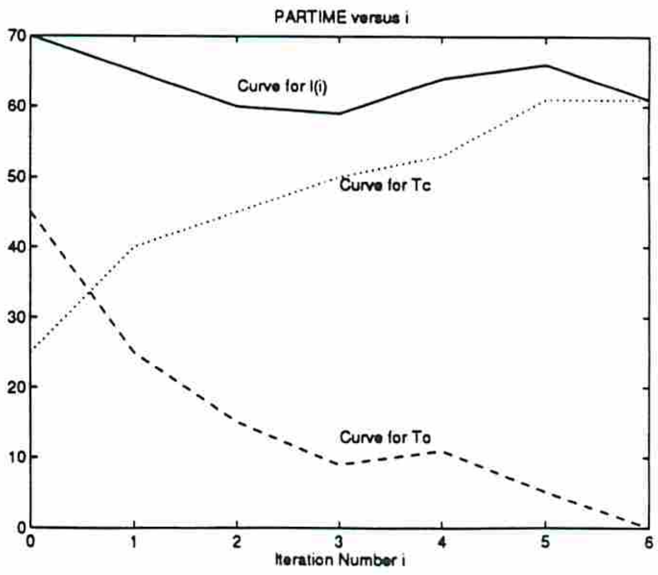


Figure 6.27: PARTIME Plot for Example 7

i	T_c	T_o	l
0	13	20	33
1	15	15	30
2	20	9	29
3	26	8	34
4	30	7	37
5	33	5	38
6	33	0	33

Table 6.2: Table for Example 1

i	T_c	T_o	l
0	25	45	70
1	40	25	65
2	45	30	75
3	60	20	80
4	65	25	90
5	75	15	90
6	75	0	75

Table 6.3: Table for Example 2

i	T_c	T_o	l
0	25	45	70
1	40	25	65
2	45	30	75
3	50	20	70
4	51	20	71
5	52	15	67
6	52	0	52

Table 6.4: Table for Example 3

i	T_c	T_o	l
0	25	45	70
1	40	25	65
2	45	30	75
3	50	20	70
4	52	20	72
5	73	1	74
6	73	0	73

Table 6.5: Table for Example 4

3. Consider the fork DAG shown in figure 6.9 (c). Here $N = 6$. The values of T_c, T_o and $l(i)$ for each iteration of the algorithm are shown in table 6.4.

   The plots for PARTIME, T_c and T_o are shown in figure 6.23.

4. Consider the fork DAG shown in figure 6.9 (d). Here $N = 6$. The values of T_c, T_o and $l(i)$ for each iteration of the algorithm are shown in table 6.5.

   The plots for PARTIME, T_c and T_o are shown in figure 6.24.

5. Consider the fork DAG shown in figure 6.9 (e). Here $N = 6$. The values of T_c, T_o and $l(i)$ for each iteration of the algorithm are shown in table 6.6.

   The plots for PARTIME, T_c and T_o are shown in figure 6.25.

6. Consider the fork DAG shown in figure 6.9 (f). Here $N = 6$. The values of T_c, T_o and $l(i)$ for each iteration of the algorithm are shown in table 6.7.

i	T_c	T_o	l
0	35	34	69
1	50	35	85
2	60	20	80
3	63	5	68
4	64	6	70
5	70	1	71
6	70	0	70

Table 6.6: Table for Example 5

i	T_c	T_o	l
0	25	45	70
1	40	25	65
2	50	20	70
3	55	10	65
4	56	5	61
5	61	1	62
6	61	0	61

Table 6.7: Table for Example 6

i	T_c	T_o	l
0	25	45	70
1	40	25	65
2	45	15	60
3	50	9	59
4	53	11	64
5	61	5	66
6	61	0	61

Table 6.8: Table for Example 7

The plots for PARTIME, T_c and T_o are shown in figure 6.26.

7. Consider the fork DAG shown in figure 6.9 (g). Here $N = 6$. The values of T_c, T_o and $l(i)$ for each iteration of the algorithm are shown in table 6.8.

   The plots for PARTIME, T_c and T_o are shown in figure 6.27.

It is easy to see that for the join DAGs in figure 6.9 (c), (e) and (f), if we don't accept the mergers that result into a higher PARTIME, we can never reach the optimal partition. In other words, the only way to reach the optimal partition is to keep merging the tasks chosen until the coarsest partition is reached.

Chapter 7

Conclusions and Future Research

7.1 Conclusions

In this thesis, we presented heuristics for automatic program code partitioning and grain size determination for DMMs. Like most partitioning methods, our approach is compile-time. Given a weighted non-hierarchical (flat) Directed Acyclic Graph (DAG) representation of the program, we proposed a data-flow based partitioning method where all levels of parallelism available in the DAG are exploited. Our procedure automatically determines the granularity of parallelism by partitioning the graph into tasks to be scheduled on the DMM. The granularity of parallelism depends only on the program to be executed and on the target machine parameters. The output of our algorithm is passed on as input to the scheduling phase.

We use the definition of code partitioning given by Sarkar [48]. Mainly, we assume the availability of an infinite number of processing elements and that the communication overhead between the processing elements is minimum but not zero, then we find the optimal partition (i.e. the one that results into minimal parallel execution time). In this case, the parallel execution time of the input program graph is the same as the CPL of the corresponding task graph. Our scheme is based on minimizing the CPL of the task graph, by performing a sequence of task merging. Since we assume that our execution model obeys the convexity constraint, our method guarantees that the task graph remains acyclic at all times, in order to avoid deadlock situations. The algorithm consists of a loop, and during each iteration of the loop a pair of tasks is chosen to be merged[1] using some heuristic. Hence, the main work of the algorithm is to decide on which tasks are to be chosen for merger during each merging iteration, with the goal being the minimization of the CPL of the task graph.

In order to come up with the criteria for task merging to be used by the heuristics, we did some analysis of the task graph to better understand the effect of merging tasks on the CPL and on the available parallelism in the Task graph. We also studied the effect of parallelism loss due to task merging on the CPL of the task graph. We determined a necessary and sufficient condition for parallelism loss as a result of task merging. Then we presented some rules to determine the mergers that result into the maximum decrease in the CPL, the mergers that result in no increase in the CPL, etc. We showed that the pair of tasks to be merged has to belong to a critical path of the task graph, otherwise the CPL can never decrease as a result of the merger. Finally, we showed that if there is no parallelism loss as a result of the merger, then the CPL of the task graph is guaranty not to increase. However, if there is parallelism loss, then the CPL could increase as a result of the merger.

Finding an optimal solution to the code partitioning problem is NP-complete. Due to the high cost of graph algorithms[2], it is nearly impossible to come up with close to optimal solutions that don't have very high cost (higher order polynomial). For instance, some of the criteria that can be used in choosing the pair of tasks to be merged, and that result into good performance in minimizing the CPL, have a very high time complexity. Therefore, we had to use criteria that result into lower performance and that have lower time complexities. In other words, we had to trade performance for less time complexity. Hence, we proposed heuristics that give reasonably good performance and that have relatively low cost. Our algorithm has a worst case time complexity of $O(E.N^3)$. However as was explained earlier in this thesis, for real applications, the average time complexity is expected to be

[1]These are called merging iterations.

[2]For our analysis we had to use various DAG traversal algorithms.

$O(N(E+N))$. For fork and join DAGs, our algorithm gives optimal performance. For complete binary trees, the performance is more than 50% of the optimal one.

7.2 Future Research

We need to do more experiments using other kinds of regular DAGs and real life benchmarks to further test our heuristics and improve them. Also, an interesting future research would be to consider complete binary trees where not all actors have the same weights and not all edges have the same weights.

Furthermore, we mentioned previously that we over-estimated the worst case time complexity of our partitioning algorithm (the complexity given is an upper bound and can never be achieved). It would be interesting to determine a tighter worst case time complexity. In other words we would like to find an achievable worst case time complexity.

In addition, if we can find ways to reduce the time complexity of our partitioning algorithm, our proposed procedure will be even more useful. Following are some methods that could be used to achieve this goal and optimize our proposed algorithm:

- Use *incremental* methods to determine CPL, ParSet(n), DepSet(n), perfect edges, safe edges, instead of recomputing them for each iteration of the algorithm. For instance, we could try to express the CPL at step n as a function of the CPL at step $n-1$ and the way the merger is done at step n. Tao Yang [18, 58, 59] uses an incremental way to determine the CPL of a task graph. He defines the *tlevel* and *blevel* of a node n in a DAG to be the length of the longest path from an entry node to n, excluding the weight of n, and the length of the longest path from n to an exit node respectively. Then he defines the priority PRIO(n) of the node n to be $tlevel(n) + blevel(n)$. It is easy to see that the node with the highest priority belongs to the critical path. Using these definitions, Tao Yang was able to make the choice of the edge to be zeroed (i.e. merged) without having to compute the critical path of the task graph, and in an incremental manner. It would be interesting to see if we can use a method similar to his, to determine the edge to be merged in our partitioning algorithm.

- Find optimal solution for *restricted classes* of DAGs.
  We expect that the time complexity of our algorithm becomes much lower when we restrict ourselves to special classes of DAGs. For instance for DAGs for which the nodes have at most one output edge, there are some properties that could enable us to find the edge to be merged in a much cheaper way. Also, we can study DAGs for which each node has at most one input edge (trees).

- In our partitioning algorithm, we keep merging tasks until we reach the coarsest partition (i.e. the partition consisting of a single task). We accept the merger even if it results in an increase in PARTIME. If we can find a way to stop the merging process much earlier without sacrificing performance, then we will have a reduction in the time complexity of the algorithm without losing any performance.

Furthermore, an interesting question to answer would be: if heuristic H_1 gives larger improvements between successive merging iterations than heuristic H_2, does that mean that H_1 performs better than H_2 ?

Finally, an interesting future research would be to investigate merging more than 2 nodes in the task graph (i.e. tasks) at a time.

Bibliography

[1] W. B. Ackerman. Efficient implementation of applicative languages. PHD Dissertation MIT/LCS/TR-323, Massachusetts Institute of Technology, Cambridge, Massachusetts, Mar 1984.

[2] Stephen J. Allan and R. R. Oldehoeft. Hep sisal: parallel functional programming. In J. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and its Applications*, pages 123–150. MIT Press, Cambridge, MA, 1985.

[3] Jean-Loup Baer. *Computer Systems Architecture.* Digital System Design. Computer Science Press, Inc., 11 Taft Court, Rockville, Maryland 20850, 1980.

[4] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic program parallelization. In *Proceedings of the IEEE*, pages 211–243. IEEE, Feb 1993. Vol. 81, No. 2.

[5] Jeffery M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, Sep 1978.

[6] D. C. Cann. Vectorization of an applicative language: Current results and future directions. In *Compcon 91*, pages 396–402, Feb 1991. Also available as a Technical Report number UCRL-JC-105654, Lawrence Livermore National Laboratory, November 1990.

[7] David Cann. Retire fortran? a debate rekindled. Technical Report UCRL-107018, Lawrence Livermore National Laboratory, Livermore, CA 94550, Jul 1991. Rev. 1.

[8] David Cann and R. R. Oldehoeft. A guide to the optimizing sisal compiler. Preprint of a paper intended for publication UCRL-MA-108369, Lawrence Livermore National Laboratory, Livermore, CA 94550, Sep 1991.

[9] David C. Cann. Compilation techniques for high performance applicative computation. PHD Thesis CS-89-108, Colorado State University, May 1989.

[10] David C. Cann, Ching-Cheng Lee, R. R. Oldehoeft, and S. K. Skedzielewski. Sisal multiprocessing support. Technical report, Lawrence Livermore National Lab, Livermore, CA 94550, 1987.

[11] Keith D. Cooper, Mary W. Hall, Robert T. Hood, Ken Kennedy, Kathryn S. McKinley, John M. Mellor-Crummey, Linda Torczon, and Scott K. Warren. The parascope parallel programming environment. In *Proceedings of the IEEE*, pages 244–263. IEEE, Feb 1993. Vol. 81, No. 2.

[12] P. Crooks and R. H. Perrott. *Language Constructs for Data Partitioning and Distribution.* The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland. email: p.crooks@v2.qub.ac.uk, r.perrott@v2.qub.ac.uk.

[13] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. Tam - a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, July 1993.

[14] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Forth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, SIGPLAN NOTICES, ACM, 11 W. 42nd St., New York, NY 10036, April 1991. ACM.

[15] Eliezer Dekel and Sartaj Sahni. Binary trees and parallel scheduling algorithms. *IEEE Transactions on Computers*, 32(3):307–315, March 1983.

[16] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the sisal language project. Technical Report UCRL-102440, Lawrence Livermore National Laboratory, Livermore, CA 94550, Jul 1990. Rev. 1.

[17] J-L. Gaudiot and L. Lee. Occamflow: A methodology for programming multiprocessor systems. *Journal of Parallel and Distributed Computing*, August 1989.

[18] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[19] D. H. Grit. A distributed memory implementation of sisal. In *Proceedings of the Fifth Distributed Memory Computing Conference*, Apr 1990. Extended Abstract.

[20] Dale H. Grit. A distributed memory implementation of sisal. Technical report, Lawrence Livermore National Laboratory, Livermore, CA 94550, Oct 1993.

[21] Matt Haines and Wim Bohm. Towards a distributed memory implementation of sisal. Technical Report CS-91-123, Colorado State University, Computer Science Department, Colorado State University, Fort Collins, CO 80523, Nov 1991.

[22] Matthew Haines and Wim Bohm. A comparison of explicit and implicit programming styles for distributed memory multiprocessors. Technical Report CS-93-104, Colorado State University, March 1993.

[23] Matthew Dennis Haines. Distributed runtime support for task and data management. PHD Dissertation CS-93-110, Colorado State University, August 1993.

[24] John P. Hayes. *Computer Architecture and Organization.* Computer Organization and Architecture. McGraw-Hill, Inc., second edition, 1988.

[25] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.

[26] S. Hiranandani, K. Kennedy, and C. W. Tseng. Compiling fortran d for mimid distributed-memory machines. *CACM*, 35(8):66–80, Aug 1992.

[27] Susan F. Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring, a method for scheduling parallel loops. *CACM*, 35(8):90–101, Aug 1992.

[28] Kai Hwang and Faye A. Briggs. *Computer Architecture and Parallel Processing.* Computer Organization and Architecture. McGraw-Hill, Inc., 1984.

[29] C. Lee. Experience of implementing applicative parallelism on cray x-mp. In *Proc. CONPAR '88*, pages 19–25, Manchester, England, Sep 1988. British Computer Society. Technical Report UCRL-98303, Lawrence Livermore National Laboratory, May 1988.

[30] C. Lee, S. K. Skedzielewski, and J. T. Feo. On the implementation of applicative languages on shared-memory, mimd multiprocessors. In *Proceedings of the Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, pages 188–197, New Haven, CT, Sep 1988. Available in SIGPLAN Notices 23,9.

[31] Liang-Teh Lee. *Occamflow: Programming a Multiprocessor System in a High-Level Data-Flow Language.* PhD thesis, University of Southern California, August 1989.

[32] E. S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, Jan 1969.

[33] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and Iteration in a Single Assignment Language: Language Reference Manual, Version 1.2, Manual M-146, Rev. 1.* Lawrence Livermore National Laboratory, Livermore, CA 94550, Mar 1985.

[34] Dan I. Moldovan. *Modern Parallel Processing.* Department of Electrical Engineering Systems, University of Southern California, Los Angeles, CA 90089-0871, 1986.

[35] R. R. Oldehoeft and S. J. Allan. Execution support for hep sisal. In J. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*, pages 151–180. MIT Press, Cambridge, MA, 1985.

[36] R. R. Oldehoeft and D. C. Cann. Applicative parallelism on a shared-memory multiprocessor. *IEEE Software*, pages 62–70, Jan 1988.

[37] R. R. Oldehoeft, D. C. Cann, and S. J. Allan. Sisal: Initial mimd performance results. In *Proceedings of the 1986 Conference on Algorithms and Hardware for Parallel Processing*, pages 120–127, Aachen, Federal Republic of Germany, Sep 1986.

[38] D. Padua, D. Kuck, and D. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, Sep 1980.

[39] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec 1986.

[40] Cherri M. Pancake. Multithreaded languages for scientific and technical computing. In *Proceedings of the IEEE*, pages 288–304. IEEE, Feb 1993. Vol. 81, No. 2.

[41] Santosh S. Pande, Dharma P. Agrawal, and Jon Mauney. Mapping functional parallelism on distributed memory machines. In John T. Feo, Christopher Frerking, and Patrick J. Miller, editors, *Proceedings of the Second Sisal Users Conference*, pages 139–159, Livermore, CA 94550, Dec 1992. Lawrence Livermore National Laboratory.

[42] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, Dec 1987.

[43] William Pugh. A practical algorithm for exact array dependence analysis. *CACM*, 35(8):102–114, Aug 1992.

[44] John E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD thesis, University of California, Davis, Livermore, CA 94550, Nov 1987. Report number UCRL-53832, Lawrence Livermore National Laboratory.

[45] Anurag Sah. Parallel language support on shared memory multiprocessors. Technical Report UCB/CSD 91/NUMBER#631, University of California Berkeley, Berkeley, CA 94720, May 1991.

[46] V. Sarkar and J. Hennessey. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the SIGPLAN 1986 Symposium on Compiler Construction*, pages 17–26, Palo Alto, CA, Jun 1986. ACM.

[47] V. Sarkar and J. Hennessey. Partioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM Conference on Lisp and functional programming,*, pages 202–211, Aug 1986.

[48] Vivek Sarkar. Partitioning and scheduling parallel programs for execution on multiprocessors. PHD Thesis CSL-TR-87-328, Stanford University, Stanford, CA 94305-2192, Apr 1987.

[49] Vivek Sarkar, Stephen Skedzielewski, and Patrick Miller. An automatically partitioning compiler for sisal. Technical Report UCRL-98289, Lawrence Livermore National Laboratory, Livermore, CA 94550, Dec 1988.

[50] S. K. Skedzielewski and M. L. Welcome. Data flow graph optimization in if1. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 17–34. Springer-Verlag, New York, NY, Sep 1985. Also in: Lecture Notes in Computer Science Number 210 (Functional Programming Languages and Computer Architecture), Springer-Verlag, Berlin, pp. 17-34, 1985. Also as a Technical Report, Lawrence Livermore National Laboratory, number UCRL-92122, Rev. 1, December 2, 1987.

[51] Stephen Skedzielewski and John Glauert. *IF1: An Intermediate Form for Applicative languages, Version 1.0, Manual M-170.* Lawrence Livermore National Laboratory, Livermore, CA 94550, Jul 1985.

[52] Stephen Skedzielewski and Robert Kim Yates. Fibre: An external format for sisal and if1 data objects, version 1.1, revision 1. Technical Report M-154, Lawrence Livermore National Laboratory, Livermore, CA 94550, Apr 1988.

[53] Stephen K. Skedzielewski and Rea J. Simpson. A simple method to remove reference counting in applicative programs. In *Proceedings ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, Jun 1989. ACM. Also available as a Technical Report

number UCRL-100156, Lawrence Livermore National Laboratory, November 23, 1988.

[54] Michael Welcome, Stephen Skedzielewski, Robert Kim Yates, and John Ranelletti. *IF2: An Applicative Language Intermediate Form with Explicit Memory Management, Manual M-195.* Lawrence Livermore National Laboratory, Livermore, CA 94550, Dec 1986.

[55] Paul G. Whiting. Compilation of a functional programming language for the csirac ii dataflow computer. Master's Thesis TR DB-91-11, Commonwealth Scientific and Industrial Research Organization, CSIRO, Division of Information Technology, 723 Swanston Street, Carlton, 3053, Australia, Oct 1991. Version 1.0.

[56] R. Wolski, J. Feo, and D. C. Cann. A prototype functional language implementation for hierarchical-memory architectures. Technical Report UCRL-JC-107437, Lawrence Livermore National Laboratory, Livermore, CA 94550, Jun 1991.

[57] Richard Wolski and John Feo. Program partitioning for numa multiprocessor computer systems. In John T. Feo, Christopher Frerking, and Patrick J. Miller, editors, *Proceedings of the Second Sisal Users Conference*, pages 111–137, Livermore, CA 94550, Dec 1992. Lawrence Livermore National Laboratory.

[58] Tao Yang. *Scheduling and Code Generation for Parallel Architectures.* PhD thesis, Rutgers University, New Brunswick, Jew Jersey, May 1993.

[59] Tao Yang and Apostolos Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

[60] Hans P. Zima and Babara Mary Chapman. Compiling for distributed-memory systems. In *Proceedings of the IEEE*, pages 264–287. IEEE, Feb 1993. Vol. 81, No. 2.