SCHEDULING AND CODE GENERATION FOR PARALLEL ARCHITECTURES

BY TAO YANG

A dissertation submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Graduate Program in Computer Science

Written under the direction of
Professor Apostolos Gerasoulis
and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey
May, 1993

ABSTRACT OF THE DISSERTATION

Scheduling and Code Generation for Parallel Architectures

by Tao Yang, Ph.D.

Dissertation Director: Professor Apostolos Gerasoulis

Automatic partitioning, scheduling and code generation are of major importance in the development of compilers for massively parallel architectures. In this thesis we consider these problems, propose efficient algorithms and analyze their performances for automatic scheduling and code generation.

In the first part of this thesis, we consider compile-time static scheduling when communication overhead is not negligible. We provide a new quantitative analysis of granularity issues to identify the impact of partitioning on optimal scheduling. We propose a new algorithm for scheduling on an unbounded number of processors named DSC, which outperforms existing algorithms in both complexity and performance. Furthermore, we study algorithms for scheduling on a bounded number of processors based on a multistep approach. We use DSC to cluster tasks and then use a load balancing and physical mapping heuristic to map the clusters onto processors. Finally we introduce a new task ordering algorithm that tries to minimize parallel time and overlap communication with computation. What distinguishes our approach with that of others in the literature is the low complexity and very good performance. We present theoretical and experimental results to verify the performance of these algorithms.

In the second part of the thesis, we consider the code generation problem for message passing architectures. We propose a new optimized method for code generation in executing a schedule of an arbitrary task graph based on an asynchronous communication model. We optimize the code by reducing communication overhead, eliminating redundant communication and improving memory utilization. We present a correctness analysis of the generated code to assure data coherence and deadlock-free communication. We also present a new multicasting algorithm for a hypercube that eliminates unnecessary routing processors.

We have implemented a compiler tool system named PYRROS that integrates the above algorithms for scheduling and code generations. The input of this system is a C program with annotated dependence information and the output is optimized parallel C code for nCUBE-I, nCUBE-II and iPSC/860 machines. The experimental results show that the performance of automatically produced code is comparable with that of hand-written code.

Acknowledgements

It is a pleasure to finally thank every one who helped me in various ways to complete my dissertation. In particular, I would like to thank my advisor, Professor Apostolos Gerasoulis, for introducing me to parallel computing and for always being there for me. His guidance and advice had a major impact on my development as a researcher and as an individual. I would also like to thank the other members of my thesis committee Professors Miles Murdocca, Constantine Polychronopoulos, Barbara Ryder and Joel Saltz for their time, support and valuable comments.

I thank Professor Michel Cosnard, Dr. Vivek Sarkar and Professor Min-You Wu for their support and comments on this work, Dr. Izzy Nelken for helping me during my initial research, Sesh Venugopal, Alain Darte, Vipul Gupta, Ajay Bakre, and Souripriya Das for useful discussion, Milind Deshpande for programming the X window schedule displayer, Probal Bhattacharjya for programming the sparse matrix graph generator and Ye Li for programming the Intel communication routines.

I also thank my friends who helped me in different ways during my stay at Rutgers: Leiguang Gong, Fan Jiang, Yong-Fong Lee, Valentine Rolfe, Jianning Song, Liping Sun, Sizheng Wei, Xiangxiang Zhang and many others.

Especially, my appreciation goes to my wife, Weining Wang, for her love and support. She provided various comments on my thesis work, programmed a random graph generator and wrote the initial version of PYRROS language parser. My appreciation also goes to my parents and other family members for their love and encouragement.

This work was partially supported by Sophia Shen Fellowship, Rutgers Graduate Fellowship and a grant No. DMS-8706122 from NSF. I thank the Department of Computer Science for providing excellent computing facilities and CAIP at Rutgers for giving us access to their nCUBE-II parallel machine.

Dedication

This thesis is dedicated to my wife and my parents.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DAG	Directed acyclic graph
CP	Critical path
DS	Dominant sequence
PT	Parallel time
SPMD	Single program multiple data
GJ	Gauss-Jordan
CD	Cholesky decomposition
GE	Gaussian elimination
LU	LU decomposition
PDE	Partial-differential equation
α	The startup time for sending a message
β	The transmission speed
p	Number of processors
v	Number of tasks
e	Number of edges
$PRED(n_x)$	Predecessors of task n_x
$SUCC(n_x)$	Successors of n_x
$g(G)$	Granularity of DAG G
$PA(n_x)$	Processor assignment for task n_x.
$ST(n_x)$	Execution starting time of n_x.
$CT(n_x)$	Completion time of n_x.

Chapter 1

Introduction

Recently, several new parallel architectures have been introduced based on the MIMD distributed memory paradigm of computation. Those include the CM-5 from Thinking Machines, the PARAGON from INTEL and the nCUBE-II hypercube. The general belief is that such architectures will be the first to achieve a teraflop performance. Unfortunately, programming such architectures, so that they are fully utilized, appears to be the major stumbling block for their widespread use. The difficulty lies in the fact that both data and programs must be partitioned and then distributed onto processors in an optimal way for efficient execution.

There are three generally distinct ways in addressing the programming difficulties for parallel architectures:

The first approach considers the problem of automatic parallelization and scheduling from sequential programs. The emphasis has been in the development of compilers or software tools that will assist in programming parallel architectures [3, 80]. One of the obstacles in the development of parallelizing compilers is the automatic identification of embedded parallelism in a sequential program. This is because the precise dependence analysis problem is not tractable in a polynomial time. Significant progress has been made towards finding approximate solutions [83, 95]. However, the false dependencies in approximated solutions could have a negative impact on performance. This was recently demonstrated by a study on the performance of automatic parallelization compilers [17], where it was found that "automatic tools produce insufficient performance improvement" due to false dependencies. In [17], it is suggested that semi-automatic tools might be more appropriate for parallelism detection.

The second approach allows a user to specify parallelism information as an input and

then a compiler system generates efficient code [14, 16, 34, 86, 94, 106]. For example, KALI [61] and FORTRAN D [51] both provide parallel constructs in their language. Even if all parallelism is available, program scheduling and communication optimization are crucial to ensure good performance of automatically produced code.

The third is a manual approach in the sense that it uses existing sequential languages with an extension of synchronization and communication library functions to distribute data and programs and schedule their execution, Ortega [77]. This approach has been the most successful with respect to the utilization of message passing architectures in the literature.

As far as we know no automatic parallelization compiler exists that can produce code for message passing machines which is as competitive to manually written code because of the difficulties in precise dependence analysis and scheduling. Programming message passing architectures manually, however, is tedious and debugging parallel programs is extremely difficult.

In this thesis we will investigate the problem of automatic scheduling and code generation for static task parallelism. More specifically, given a precedence graph description of task parallelism, is it possible to automatically generate parallel code *efficiently* whose performance is on average *comparable* to that of a good hand-coded program?

This problem is important because a compiler with complexity higher than the sequential execution of the program has very limited applicability. Also an automatically generated program that is, say, a factor of 10 off a good manual program will considerably underutilize an architecture with, say, a teraflop performance.

1.1 Assumptions and Scope

This thesis considers both theory and systems issues in static scheduling and automatic program generation for parallel architectures. We use a directed acyclic task graph to model parallel computation. We do not address the automatic generation of task graphs and assume that such a graph is available at compile-time. When a task graph in this thesis is associated with program code, program segments of tasks are allowed to

access data items in a shared memory programming style. In this way, our techniques can be used whenever a graph is derived from a sequential shared memory program. The target machine is MIMD distributed memory architecture such as nCUBE-II and INTEL iPSC/860.

We briefly discuss the main research problems considered in this thesis below.

- *Scheduling*

  Given an abstract representation of a static task graph, schedule it onto multiprocessors with communication overhead. The main optimization issues are balancing computation among processors, reducing inter-processor communication and overlapping communication with computation. Most of these problems are NP-complete or NP-hard. Our goal is to design low complexity algorithms without compromising performance.

  A very successful approach to scheduling is a two-step method, e.g. Sarkar [90], Ortega [77]. At the first step a task graph is clustered assuming that there is a sufficient number of fully connected processors. When two tasks are assigned in the same cluster, they are executed in the same processor. At the second step, clusters are mapped onto physical processors. Kim and Browne [59, 60] have experimented with the clustered two-step method and the unclustered one-step scheduling method and they found that clustered scheduling results in substantial improvements in performance. Gerasoulis and Nelken [39, 74] have used this two-step method in parallel linear algebra computation on nCUBE and CYBER machines. In this thesis, we will follow this approach.

- *Code generation*

  Give a task graph and a schedule for it, generate SPMD (single program multiple data) code that executes program segments with respect to the given schedule. Various code optimization strategies need to be considered. The correctness of produced code also needs to be studied to avoid data inconsistency and deadlock.

- *Software system*

Develop a software system that integrates scheduling and code generation techniques to verify theoretical and algorithmic results in real architectures such as nCUBE-II and INTEL iPSC/860.

1.2 Contributions

The contributions of this thesis work are as follows:

- *Scheduling*

  We have developed a granularity theory to analyze the impact of partitioning on the selection of processor assignment strategies for optimal scheduling. It provides a quantitative analysis that captures the trade-off point between parallelization and sequentialization in scheduling general task graphs.

  We have developed a framework for the design and evaluation of clustering algorithms. The DSC (Dominant Sequence Clustering) algorithm is the first clustering algorithm that works for general graphs but also is optimal for special classes of DAGs and has a low complexity. Our theoretical and experimental results show that DSC outperforms the other existing clustering algorithms in terms of both complexity and parallel time.

  We have developed a framework for evaluating the classical list scheduling algorithms when communication is not considered and studied the theoretical properties that provide an explanation to the near optimal performance of the CP (critical path) algorithm. Then we extend these results to the task ordering problem when communication is present and develop a new heuristic, the RCP algorithm, with a good performance.

- *Code generation*

  We have developed a generic code generation scheme with the integration of several code optimization strategies for executing an arbitrary DAG on a message passing machine. The code optimization includes: 1) Selection of one-to-one, one-to-some, and one-to-all communications. 2) Removal of redundant communication

primitives. 3) Release of useless data space. 4) Reduction of idle communication times.

We have considered issues of data coherence and communication deadlock and presented a theoretic analysis that ensures the correctness of the generated code for executing an arbitrary DAG. We have also considered the case when a graph is executed iteratively. Finally, we have developed a new multicasting algorithm on a hypercube that eliminates the unnecessary routing overhead.

- *Software System*

  We have implemented a compile tool system named PYRROS[1] that takes C programs and their dependencies as input and produces parallel C code. The function modules of the PYRROS system are shown in Figure 1.1. The current PYRROS tool has the following components: a task graph language with an interface to C, allowing users to define partitioned programs and data; a scheduling system for clustering the graph, load balancing and physical mapping, and communication/computation ordering; a graphic displayer for displaying task graphs and scheduling results; a code generator that inserts synchronization primitives and performs code optimization for nCUBE-I, nCUBE-II and INTEL iPSC/860 hypercube machines.

  PYRROS is the first compile-time system that integrates static scheduling and code generation techniques and executes general program task graphs efficiently on real message passing machines.

1.3 Thesis Organization

The thesis organization is shown in Figure 1.2. In Chapter 2 we describe the basic terminology and demonstrate the importance of program scheduling. In Chapter 3, We also address the impact of program partitioning and granularity on scheduling. In

[1] The name PYRROS comes from PYRRHUS the legendary king of Epirus.

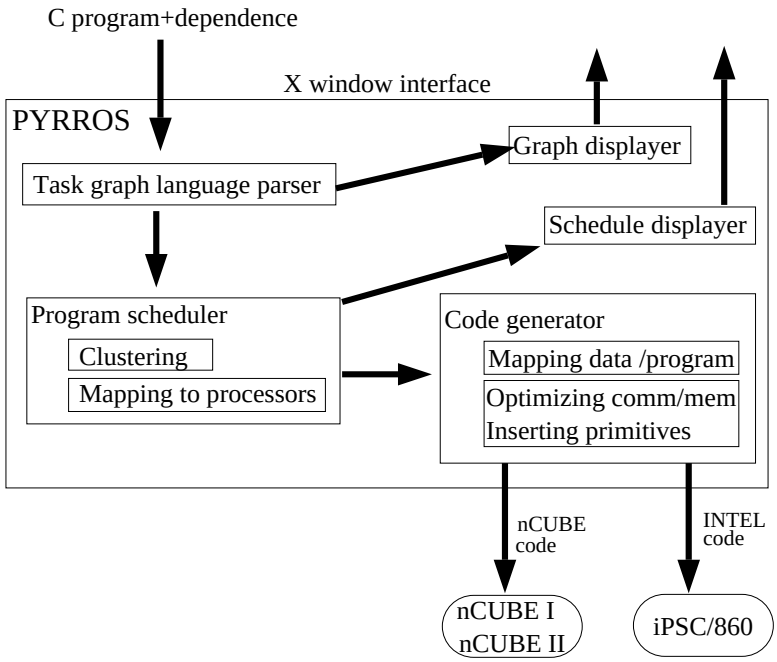


Figure 1.1: The function modules in PYRROS.

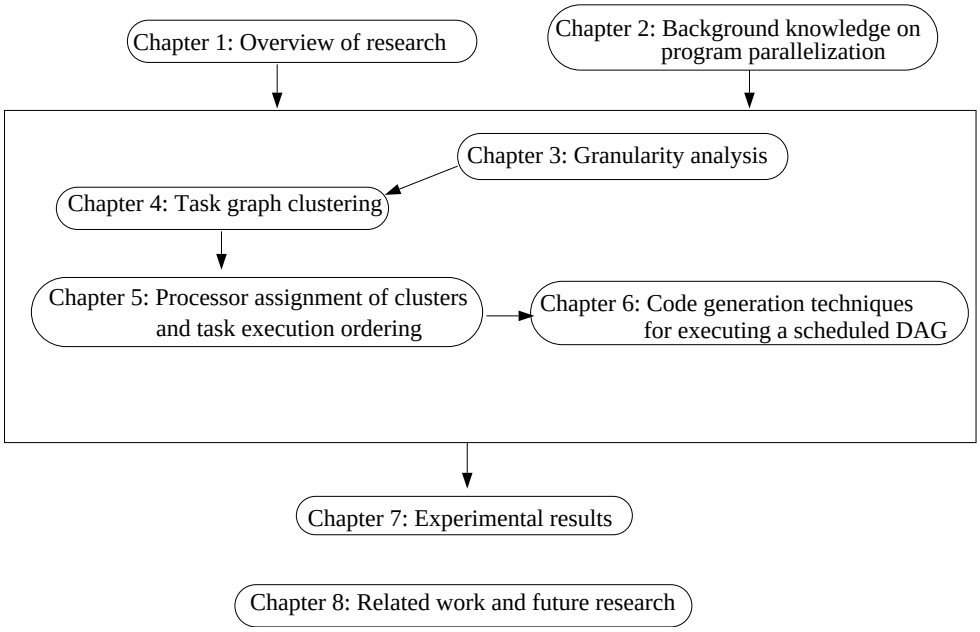


Figure 1.2: The thesis organization.

Chapter 4 we present a fast clustering algorithm called DSC. We also present a framework of clustering algorithm in order to analyze and compare clustering algorithms. In Chapter 5 we describe algorithms for mapping clustered tasks onto processors. For the task execution ordering problem, we present a framework of analyzing ordering algorithm and based on that we derive a fast algorithm called RCP. In Chapter 6 we describe algorithms for code generation. In Chapter 7 we present some experiments to show that the performance of automatically-generated code is comparable to that of hand-written code. In Chapter 8 we discuss related work and future work.

Chapter 2

Background

In this chapter, we present a brief discussion of background knowledge and a demonstration on the importance of program scheduling.

2.1 Parallel Architectures

Most of currently commercially-available parallel machines can be categorized as SIMD and MIMD. SIMD stands for single instruction multiple data, requiring all processors perform the same instruction at each clock cycle. In a MIMD architecture (multiple instructions and multiple data), different processors can perform different instructions on different data at each clock cycle.

There are two basic types of MIMD architectures, shared memory and distributed memory. In a share memory architecture, there is a global memory directly accessible by individual processors. An example of such a machine is SEQUENT. In a distributed memory machine, there is no global memory, each processor owns a local memory and processors communicate with each other through message passing. An example of such a machine is INTEL iPSC/860. Distributed memory architecture is more scalable than share memory architecture since the global memory is a bottleneck and it has been shown that it is relatively cheap to build a message-passing machine by hooking thousands of processors together without performance degradation. However, programming distributed memory machines is more difficult than for shared memory architecture because there is no concept of global memory in programming. In this thesis we focus on message passing distributed memory architectures.

An important characteristic of a multiprocessor architecture is the topology of interconnection network, i.e. how processors are connected. Examples of network are

clique, ring, linear array, hypercube, and mesh. Machines from INTEL and nCUBE use hypercube and mesh. In this thesis we assume a parallel machine has p processors with processor number as $0, 1, \cdots, p-1$. Define $Dist(p_i, p_j)$ as the distance of two processors in the interconnection network. If an architecture is fully connected (called a clique), then $Dist(p_i, p_j) = 1$.

An important factor of a machine that affects program performance is its speed in computation and communication. For inter-processor communication, a popular linear model is used to quantify transmission delay in sending a message of size N between two neighbor processors: $\alpha + N\beta$ where α is the startup time and β is the transmission rate. The communication delay between two non neighboring processors p_i and p_j is usually estimated using the linear model: $Dist(p_i, p_j)(\alpha + N\beta)$. For a machine with a wormhole mechanism, this expression overestimates the actual delay. Dunigan [32] gives tables for estimating such delays on INTEL iPSC/86 and nCUBE-II based on benchmarking. However with communication contention in a network, his formula may underestimate the actual delay. In this thesis, we use the linear model.

2.2 Program Parallelization

There are two major stages in program parallelization.

- Program partitioning and parallelism identification.

  Given a program, we group program segments into a basic unit of computation that could contain statement(s) and/or procedure(s). Each basic unit of computation is called a *task*. Then the dependence relation between tasks constitutes their communication edges. There are three basic types of data dependence: *true*, *anti* and *output*. Data dependency analysis technique can be found in Wolfe and Banerjee [95]. Control dependence [35] is not considered in this thesis.

  Tasks and their dependence constitute a direct acyclic dataflow graph (DAG). We can assign weights to nodes and edges. A node weight is the time for executing this computation unit. An edge (n_i, n_j) is associated with a data item that is produced in n_i and used in n_j. Its edge weight is the *shortest* time for transferring this data

item between two processors if n_i and n_j are in separate processors. The compiler algorithms for deriving the average weight values are discussed in Sarkar [91].

We use the *macro-dataflow task model* of execution. A task receives all input in parallel before starting execution, executes to completion without interruption, and immediately sends the output to all successor tasks in parallel, El-Rewini and Lewis [33], Sarkar [90], Wu and Gajski [98].

- Scheduling

  In this stage, program tasks and associated data items are mapped onto processors. It is called *processor assignment*. Within each processor, tasks are executed one by one following some order. Message receiving and sending is conducted before and after a task execution if inter-processor communication is necessary.

We will demonstrate the process of program parallelization in next section. Now we give a formal definition of a DAG and a schedule. A weighted DAG is a tuple $G = (V, E, \mathcal{C}, \mathcal{T})$, where

- $V = \{n_j, j = 1 : v\}$ is the set of nodes and $v = |V|$.
- $E = \{e_{i,j} = (n_i, n_j)\}$ is the set of communication edges and $e = |E|$.
- $\mathcal{C}$ is the set of edge communication costs. The value $c_{i,j} \in \mathcal{C}$ is the communication cost incurred along the edge $e_{i,j} \in E$, which is zero if both nodes are mapped in the same processor.
- $\mathcal{T}$ is the set of node computation costs. The value $\tau_i \in \mathcal{T}$ is the computation cost for node $n_i \in V$.
- The *width* of a DAG is the size of the maximal set of independent tasks. It is also called *degree* of parallelism of this DAG.
- $PRED(n_x)$ is the set of predecessors of task n_x.
- $SUCC(n_x)$ is the set of successors of task n_x.

An example of DAG is shown in Figure 2.1(a) with 8 tasks $n_1, n_2, \cdots, n_8$. Their execution times are in the right side of the bullets and edge weights are written in the edges.

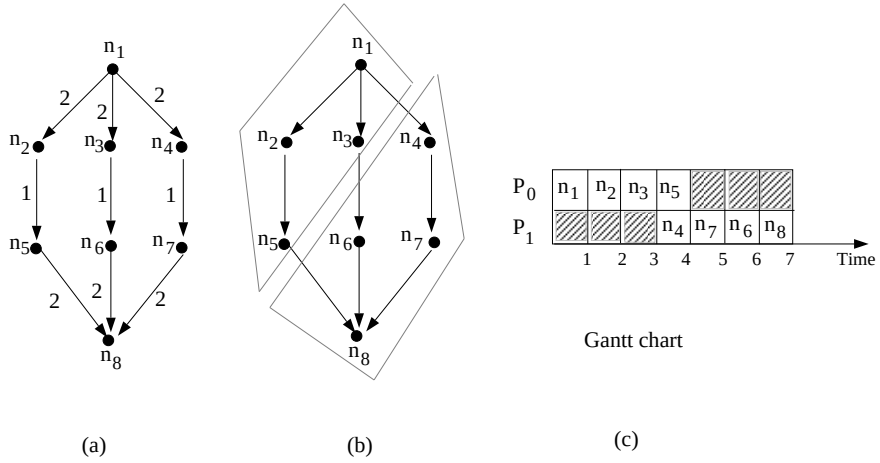


Figure 2.1: (a) is a DAG, node weights are 1. (b) is a processor assignment of nodes. (c) is the Gantt chart of a schedule.

Given a DAG and p processors, a schedule is a mapping of tasks to processors and the assignment of execution starting time for each task. We define:

- $PA(n_x)$ is the processor number for task n_x.
- $ST(n_x)$ is the execution starting time of n_x.
- $CT(n_x)$ is the completion time of n_x.

For any edge (n_x, n_y), if the two tasks are mapped into the same processor, i.e., $PA(n_x) = PA(n_y)$, edge weight $c_{x,y}$ between two tasks is zero assuming that the fetching of data from local memory costs significantly less than that from remote off-processor sites. When two tasks are mapped onto different processors, $c_{x,y}$ is adjusted according to the physical distance.

A schedule is said *valid* if the execution ordering satisfies the precedence constraints. The parallel time of a schedule is $PT = max_{i=1}^{v} CT(n_i)$. A schedule is said *optimal* if it gives the minimum parallel time.

Figure 2.1(b) is a processor assignment of tasks in (a) and (c) is Gantt chart representation of a valid scheduling solution.

2.3 An Example of Program Parallelization

We demonstrate program parallelization and the importance of program scheduling using the following simple program.

```
x(1)=1;
for i = 2 to 4
    x(i)=x(1);
end
for i = 1 to 2
    y(i+1)=2*x(i);
end
```

We assume each assignment statement to be a task and the partitioned program is shown below:

```
S1: x(1)=1;
    for i = 2 to 4
S2:         x(i)=x(1);
    end
    for i = 1 to 2
S3:         y(i+1)=2*x(i);
    end
```

The dependence task graph is shown in Figure 2.2(a) where task node S_2^i stands for the i-th loop iteration in executing statement S_2. We assume that the computation time of S_1 and S_2 is 1 time unit and S_3 is 2 time units. The transmission delay between two processors for one data element takes 4 time units. We mark the computation weight in the nodes and the communication delay in the edges in Figure 2.2(a).

We list three scheduling solutions. The parallel time is computed assuming that processors are fully connected.

- Solution 1: nonoptimal

  A nonoptimal solution is depicted in Figure 2.2(b). The parallel time(PT) of executing this task graph is equal to 12. This processor assignment can be produced by Fortran D [51]. Their approach to program mapping is to let a user specify

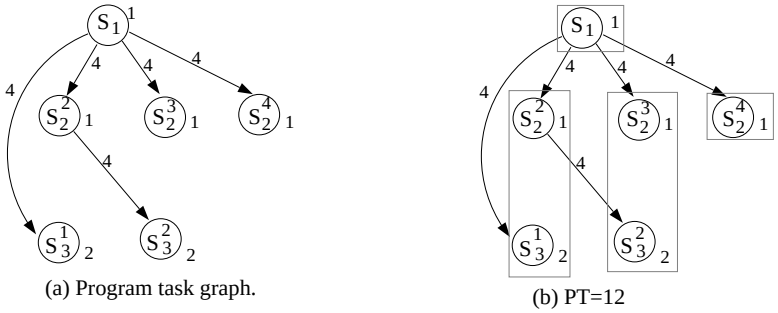


Figure 2.2: (a) An example of task graph. (b) An unoptimized mapping.

the distribution of data items then the system will assign program statements to processors based on so called "owner computes rule" [14, 86]: a processor executes a computation unit if this unit modifies the data that the processor owns.

Assume that a user distributes array x and y according to the element indices, i.e. processor i owns $x(i)$ and $y(i)$. Then the owner computes rule implies the following processor assignment of tasks: $PA(S_1) = 0$ and $PA(S_2^i) = i - 1$ and $PA(S_3^i) = i$. This results in the assignment shown in Figure 2.2(b).

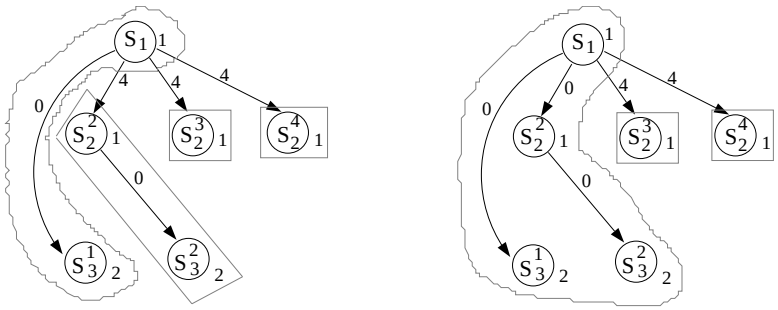
- Solution 2: an improved version

  Another solution is shown in Figure 2.3(a) the PT is 8 which is better than solution 1. This solution can be produced in Fortran D by using *index alignment*: A user can improve the previous processor mapping by analyzing the relationship between array x and y in the statement S_3. By changing the distribution of array y so that processor i owns $x(i)$ and $y(i+1)$ then by the owner computes rule the new processor assignment is shown in Figure 2.3(a). The weights of communication edge (S_1, S_3^1) and (S_2^2, S_3^2) become zero and the parallel time is reduced to 8.

- Solution 3: optimal

  A better solution can be obtained by a program scheduling algorithm that utilizes the computation and communication weight information to optimize the mapping of data items and tasks so that the parallel time is minimized. Figure 2.3(b) shows a processor assignment where 3 processors are used and S_1, S_2^2 and S_3^1 and S_3^2 are

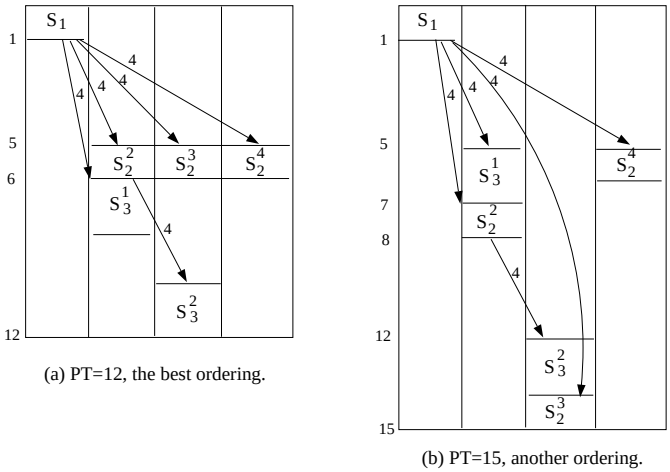
mapped into one processor. The optimized parallel time is 6.



(a) PT=8 after alignment. (b) PT=6 by program scheduling.

Figure 2.3: (a) An improved schedule. (b) The best solution.

The above example demonstrates the importance of processor assignment. Another important part in scheduling is the task ordering. Suppose we use the processor assignment shown in Figure 2.2(b), then executing S_2^2 before S_3^1 and S_2^3 before S_3^2 has the parallel time 12. But executing S_2^2 after S_3^1 and S_2^3 after S_3^2 will increase the parallel time to 15. The two orderings are depicted in Figure 2.4.



(a) PT=12, the best ordering. (b) PT=15, another ordering.

Figure 2.4: Two ordering solutions for Figure 2.2(b).

To conclude, significant performance improvement in parallel execution can be achieved by program scheduling. Automating optimizations in scheduling and code generation is important to eliminate the burden of users in scheduling and coding parallel programs.

Chapter 3

Granularity Analysis: Impact of Program Partitioning on Scheduling

In this chapter we analyze the impact of program partitioning and task graph granularity on the selection of scheduling strategies. This analysis will help in the design of scheduling and program partitioning. Basically there are two distinct strategies for scheduling: parallelizing tasks or sequentializing tasks. The trade-off point between parallelization and sequentialization is closely related to the granularity value: the ratio between the task computation and communication. If communication cost is too high, parallelization is not encouraged.

Stone [93] has analyzed the granularity issues by examining a set of tasks in which every task computes R units of time and communicates with all other tasks at an overhead of C. He defines the ratio R/C as the *task granularity* and shows that it is this ratio that determines the optimum trade-off point between parallelism and sequentialization. The weakness of this analysis is that his model does not consider task precedence and also assumes equal task weights and communication weights.

We formally study the granularity issues for general task graphs with arbitrary weights and precedence constraints. We relate parallelization with *linear clustering* and sequentialization with *nonlinear* clustering defined in next section. We propose a new definition of the granularity for a DAG which captures a trade-off point when linear clustering outperforms nonlinear clustering.

The organization of this chapter is as follows: Section 3.1 gives the definition of linear and nonlinear clustering and demonstrates the impact of granularity on scheduling. Section 3.2 proposes a formal analysis for a general DAG. Section 3.3 gives experimental results.

3.1 Preliminaries

We assume that there is a sufficient number of processors which are are fully connected. We interchangeably call a processor a cluster and vice versa in this chapter.

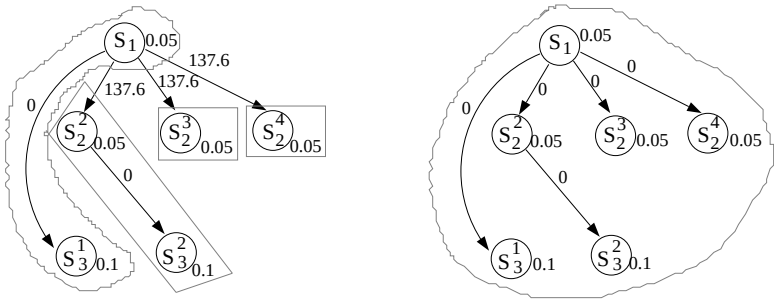
We distinguish between two kinds of clusters: *linear* and *nonlinear*. A cluster is called nonlinear if there are two independent tasks within the same cluster, otherwise is called linear. In Figure 2.3(a) there are four linear clusters. On the other hand, in Figure 2.3(b) the left cluster is nonlinear because there are no precedence relation between S_3^1 and S_3^2 and they are independent. In a nonlinear cluster, a schedule enforces the sequentialization of independent tasks and as a result this cluster is considered linearized.

Linear clustering fully exploits the parallelism while nonlinear clustering reduces the parallelism to avoid high communication cost. Next we demonstrate how the granularity impacts the choice of clustering strategies.

3.1.1 A demonstration of impact of granularity on scheduling

Example 1: In the example of Section 2.3, task partitioning is at the statement level: a simple assignment statement is a task and the message communicated between tasks is a scalar data item. This statement-level partitioning is not suitable for the current scalable architecture such as iPSC/860 and nCUBE-II since there is a large overhead in startup time for message transmission. For single precision arithmetic in iPSC/860, it takes only $0.1\mu s$ to perform an addition or multiplication while the time delay for transmitting N single precision words between two adjacent processors is $\alpha + N\beta$ where $\alpha = 136\mu s$ and $\beta = 1.6\mu s$. Assume that copying takes half of the time for addition. The computation time of statement S_3 in Figure 2.2(a) is $0.1\mu s$ and the time of S_1 and S_2 is $0.05\mu s$. The communication weights of the task graph in Figure 2.2(a) will be $137.6\mu s$ for sending a single precision floating point number. By letting processor i own both $x(i)$ and $y(i+1)$ and using the owner computes rule, the parallel time of executing this graph on iPSC/860 will be $137.8\mu s$, as shown in Figure 3.1(a). The best solution shown in Figure 3.1(b) is to assign all tasks into one processor and sequentialize their

execution, which results in PT=0.6μs.



(a) PT=137.8 on iPSC/860. (b) PT=0.6 , the best solution.

Figure 3.1: (a)Preserve the parallelism on iPSC/860. (b) Sequentialize parallel tasks.

Example 2: For the precedence DAG shown in Figure 3.2, if the computation cost w is greater or equal to communication cost c then the parallel time is minimum when n_2 and n_3 are executed in two separate processors as shown in Figure 3.2(c). In this case all parallelism in this graph can be fully exploited since it is "useful parallelism". If on the other hand we assume that $w < c$, then the parallelism is not "useful" since the minimum parallel time is derived by sequentializing the tasks n_2 and n_3 as shown in Figure 3.2(b).

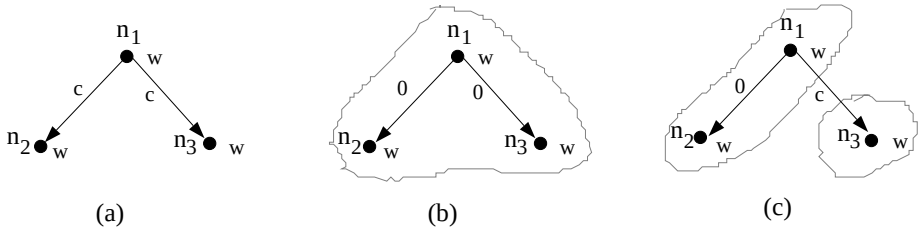


Figure 3.2: Sequentialization vs. parallelization. (a) A weighted DAG. (b) Sequentialization using non-linear clustering(c) Parallelization using a linear clustering.

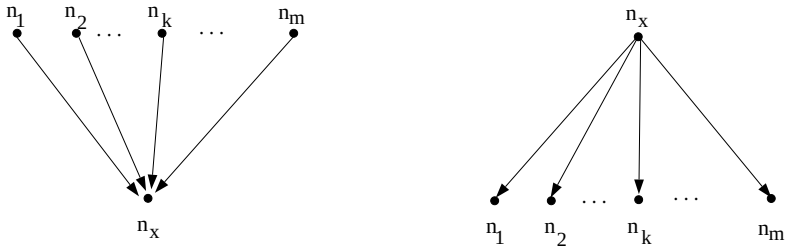
Notice that linear clustering preserves the parallelism embedded in the DAG while non-linear clustering does not. We observe the following:

> If the execution of a DAG uses linear clustering and attains the optimal time, then this indicates that the granularity of the DAG is appropriate for the given architecture; otherwise it is too fine and the scheduling algorithm still has to execute independent tasks together in the same processor using a nonlinear clustering strategy.

So far we have demonstrated the impact of the granularity on simple clustering cases. It is of interest to see if this analysis can be generalized to arbitrary DAGs.

3.2 A Formal Analysis

We first give a quantitative granularity definition for a DAG. Then we show that our granularity definition plays an important role in determining the trade off point between parallelization and sequentialization of tasks in arbitrary DAGs.



(a) Join set J_x (b) Fork set F_x

Figure 3.3: Fork and join sets.

A DAG consists of *fork* or/and *join* sets shown in Figure 3.3. The join set J_x consists of all immediate predecessors of node n_x. The fork set F_x consists of all immediate successors of node n_x. Let $J_x = \{n_1, n_2, \ldots, n_m\}$ and define

$$g(J_x) = \min_{k=1:m} \{\tau_k\} / \max_{k=1:m} \{c_{k,x}\}.$$

Similarly let $F_x = \{n_1, n_2, \ldots, n_m\}$ and define

$$g(F_x) = \min_{k=1:m} \{\tau_k\} / \max_{k=1:m} \{c_{x,k}\}.$$

We introduce the *grain* of a task as

$$g_x = \min\{g(F_x), g(J_x)\}$$

and the *granularity* of a DAG as

$$g(G) = \min_{x=1:v} \{g_x\}.$$

We call a DAG *coarse grain* if $g(G) \geq 1$ otherwise *fine grain*. If $\tau_k = R$ and $c_{i,k} = C$ then the grain of every task and the granularity of the DAG reduces to the ratio R/C

which is the same as Stone's definition. For coarse grain DAGs each task receives or sends a small amount of communication compared to that the computation of its adjacent tasks.

Consider the DAG in Figure 2.1(a) and assume that $\tau_i = 1$ for all tasks and the communication cost shown in that figure. The width of this graph is 3. The granularity of the DAG is 1/2 and therefore the DAG is not coarse grain. For the example in Figure 3.2(a), $g(G) = w/c$. When $g(G) \geq 1$, linear clustering in Figure 3.2(c)is optimal. But when $g(G) < 1$, sequentialization is needed. This example also gives some intuitive feeling on the impact of the granularity on the decision of clustering strategies. In general, we have the following result.

Lemma 3.1 *Given a coarse grain DAG G and its the reduced graph derived by deleting all transitive edges of G, the schedule for the reduced graph satisfies all precedence constraints of G.*

Proof: Suppose that there is a path $n_1 \rightarrow n_2 \rightarrow ... \rightarrow n_k$ where $k > 2$, and that there is also a transitive edge from n_1 to n_k. Next assume that the transitive edge $< n_1, n_k >$ is deleted. The task n_k still cannot be ready for execution for any schedule until the completion of all tasks n_1, n_2, ..., and n_{k-1}, that is,

$$ST(n_k) \geq CT(n_1) + \sum_{i=2}^{k-1} \tau_i.$$

Since the DAG is coarse grain then

$$c_{1,k} \leq \tau_2 \leq \sum_{i=2}^{k-1} \tau_i$$

which implies that

$$ST(n_k) \geq CT(n_1) + c_{1,k}.$$

This indicates that the precedence constraint $< n_1, n_k >$ is automatically satisfied. □

Theorem 3.1 *Given a coarse grain DAG, any nonlinear clustering of this DAG can be transformed into a linear clustering with less or equal parallel time.*

Proof: According to Lemma 3.1, we can assume that the given DAG has no transitive edges. There are two parts in the proof. The first part describes a procedure that extracts linear clusters from nonlinear clusters. The second part shows how to construct a schedule at each step of the extraction whose parallel time is less or equal to the schedule in the previous step.

Let us assume that $CLUSTER(0)$ is the set of the initial nonlinear clustering and S is its optimum schedule under the assumption that each cluster is executed on a separate processor of a clique architecture. The set $CLUSTER(0)$ consists of nonlinear NLC and linear LC clusters and there exists at least one $NLC \in CLUSTER(0)$. The optimum schedule defines execution sequence, $\{\preceq_i\}$, of the tasks for each processor i. The linearization procedure is given below:

INPUT: $CLUSTER(0)(nonlinear)$; S;

OUTPUT: $CLUSTER(l)(linear)$; S_l;

$l = 0$; $S_0 = S$;

FOR each processor i that contains a nonlinear cluster NLC **DO**

  WHILE NLC is not linear **DO**

    Extract a linear cluster from NLC and assign LC to a new processor j;

    $NLC = NLC - LC$;

    $CLUSTER(l+1) = CLUSTER(l) \cup \{LC\}$;

    Construct a schedule S_{l+1} from S_l with smaller parallel time;

    $l = l + 1$;

  ENDWHILE

ENDFOR

Define

- $PRED(n_k)$ to be the set of all immediate predecessors of n_k in G.
- $pred_l(n_k) = (n_{k_1}, \ldots, n_{k_t}, \ldots, n_{k_q})$ to be the sequence of all immediate predecessors of n_k assigned to the same processor as n_k *for schedule* S_l. This sequence is derived from the order imposed by S_l.

- $R_l(n_t) = PRED(n_t) - pred_l(n_t)$, the set of all immediate predecessors of n_t that are not in the same processor as n_t for schedule S_l. The "$-$" symbol is the set difference operation.

We first show how to extract a linear cluster from a nonlinear cluster using a bottom-up traversal of the graph and then show that it is possible to construct a schedule for the new clustering with less or equal parallel time. We show both the extraction and the construction inductively. Assume that $NLC = \{n_1, n_2, \ldots, n_m\}$ is the nonlinear cluster in processor i and that the execution order imposed by S_l is $\{\preceq_i\} = (n_1, n_2, \ldots, n_m)$.

Extract a linear cluster from a nonlinear cluster:

- Extract n_m first, by removing the node from NLC.
- Assume n_k has been extracted. Then, we show how to extract the next node. There are three cases for performing the extraction:

  Case 1: If $pred_l(n_k) = \emptyset$ then stop the extraction. The node n_k is the first node in LC.

  Case 2: If $pred_l(n_k) = (n_{k_1}, \ldots, n_{k_q})$ and there exists **no** successor n_{s_q} of n_{k_q} inserted between n_{k_q} and n_k in $\{\preceq_i\}$, then extract n_{k_q} and assign it into LC.

  Case 3: If $pred_l(n_k) = (n_{k_1}, \ldots, n_{k_q})$ and there exists a successor n_{s_q} of n_{k_q} such that

  $n_{k_q} \preceq_i n_{s_q} \preceq_i n_k$, then stop the extraction. The node n_k is the first node of LC.

Notice that the while loop assures that all linear clusters have been extracted from NLC. Also the extracting continues even if NLC becomes linear since checking of linearity of NLC is done after the extraction stops.

Define $ST_l(n_p)$ the *starting time* and $RT_l(n_p)$ the *ready time* of node n_p in S_l. The ready time is the time when n_p has received all data from its immediate predecessors and is ready for execution. We have that

$$ST_l(n_p) \geq RT_l(n_p).$$

We also need to define two functions

$$P_l(n_t, X) = \max_{n_p \in X}\{ST_l(n_p) + \tau_p\}, \qquad Q_l(n_t, X) = \max_{n_p \in X}\{ST_l(n_p) + \tau_p + c_{p,t}\}$$

for schedule S_l where X is a subset of $PRED(n_t)$.

Next we construct a schedule S_{l+1} from S_l with the same execution order, i.e. if $n_j \preceq_k n_p$ in S_l then the order will remain the same in S_{l+1} if the nodes are in the same processor. We construct S_{l+1} by topologically examining each task in G and showing that $ST_l(n_p) \geq RT_{l+1}(n_p)$. Consequently we can define S_{l+1} such that $ST_l(n_p) \geq ST_{l+1}(n_p)$ for all $n_p \in G$, which imply that the parallel times satisfy $PT_{l+1} \leq PT_l$. Assume that for schedule S_l the extracted linear cluster from $NLC = \{n_1, n_2, \ldots, n_m\}$ is $LC = \{n_{c_1}, n_{c_2} \ldots, n_{c_h}\}$ and $n_{c_h} = n_m$. We present the construction inductively:

Construct a schedule:

- First set $ST_{l+1}(n_p) = ST_l(n_p)$ for all entry nodes n_p which have no predecessors.
- Assume for a node n_t that $ST_l(n_p) \geq ST_{l+1}(n_p)$ for all $n_p \in PRED(n_t)$. We only consider schedules S_{l+1} with the same execution order as S_l. There are four distinct cases to consider.

1. n_t **is not in** NLC or LC.

Refer to Figure 3.4(1). The node n_t is in some processor $r \neq i$. We have that

$$RT_l(n_t) = \max\{P_l(n_t, pred_l(n_t)),\ Q_l(n_t, R_l(n_t))\}$$

$$RT_{l+1}(n_t) = \max\{P_{l+1}(n_t, pred_{l+1}(n_t)),\ Q_l(n_t, R_{l+1}(n_t))\}.$$

This is because each $n_p \in PRED(n_t)$ sends the *data in parallel* immediately after completion of its execution and since the architecture is a *clique* the arrival time of the data for n_t is not affected if LC becomes a new processor j.

Because the predecessors of n_t in the same processor are not changed by the extraction we have that

$$pred_{l+1}(n_t) = pred_l(n_t), \qquad R_{l+1}(n_t) = R_l(n_t).$$

The above combined with the induction hypothesis $ST_l(n_p) \geq ST_{l+1}(n_p)$ for all $n_p \in PRED(n_t)$ imply

$$P_{l+1}(n_t, pred_{l+1}(n_t)) \leq P_l(n_t, pred_l(n_t)), \quad Q_{l+1}(n_t, R_{l+1}(n_t)) \leq Q_l(n_t, R_l(n_t)).$$

Therefore

$$RT_{l+1}(n_t) \leq RT_l(n_t) \leq ST_l(n_t)$$

and because the execution order in S_{l+1} is chosen to be the same as in S_l we can always set

$$ST_l(n_t) \geq ST_{l+1}(n_t).$$

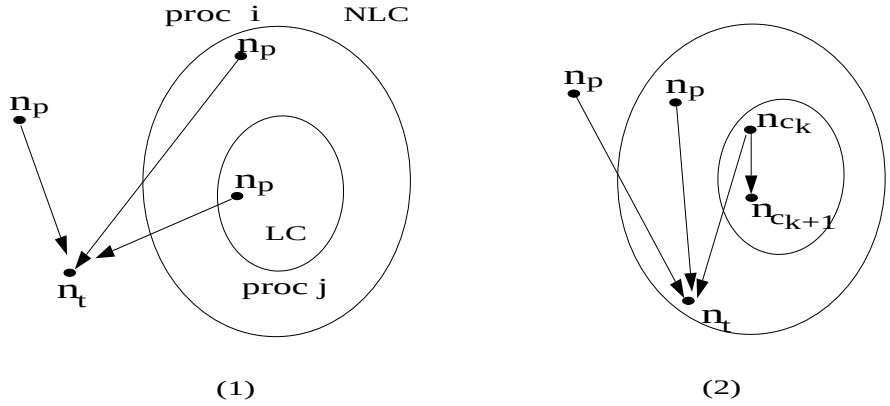


Figure 3.4: Case 1 and 2 of **construct** procedure.

2. n_t **is in** $NLC - LC$

Refer to Figure 3.4(2). Since tasks in LC are extracted from processor i, the ready time of n_t is possibly delayed when one of its immediate predecessors, say n_{c_k}, is in LC. This is because n_{c_k} is moved from processor i to processor j and the communication between n_{c_k} and n_t becomes nonzero. According to the extraction procedure, another immediate successor of n_{c_k}, say $n_{c_{k+1}}$, must be in LC and they must satisfy $n_{c_k} \preceq_i n_{c_{k+1}} \preceq_i n_t$. Therefore

$$ST_l(n_t) \geq \max\{P_l(n_t, pred_l(n_t)),\ Q_l(n_t, R_l(n_t)),\ ST_l(n_{c_{k+1}}) + \tau_{c_{k+1}}\}$$

$$RT_{l+1}(n_t) = \max\{P_{l+1}(n_t, pred_{l+1}(n_t)),\ Q_{l+1}(n_t, R_{l+1}(n_t))\}.$$

Because n_{c_k} is on a different processor at schedule S_{l+1} we have that

$$pred_{l+1}(n_t) = pred_l(n_t) - \{n_{c_k}\}, \quad R_{l+1}(n_t) = R_l(n_t) \cup \{n_{c_k}\}.$$

We can easily see from the definition of P and Q and the induction hypothesis

$$P_{l+1}(n_t, pred_l(n_t) - \{n_{c_k}\}) \leq P_l(n_t, pred_l(n_t)), \quad Q_{l+1}(n_t, R_l(n_t)) \leq Q_l(n_t, R_l(n_t))$$

$$Q_{l+1}(n_t, R_l(n_t) \cup \{n_{c_k}\}) = \max\{Q_{l+1}(n_t,\ R_l(n_t)),\ ST_{l+1}(n_{c_k}) + \tau_{c_k} + c_{c_k,t}\}.$$

We also have that

$$ST_l(n_{c_{k+1}}) \geq ST_l(n_{c_k}) + \tau_{c_k}.$$

Since the graph is coarse, then $\tau_{c_{k+1}} \geq c_{c_k,t}$ and

$$ST_l(n_{c_{k+1}}) + \tau_{c_{k+1}} \geq ST_l(n_{c_k}) + \tau_{c_k} + c_{c_k,t} \geq ST_{l+1}(n_{c_k}) + \tau_{c_k} + c_{c_k,t}.$$

The above imply that

$$ST_l(n_t) \geq RT_{l+1}(n_t)$$

and thus we can set

$$ST_{l+1}(n_t) \leq ST_l(n_t).$$

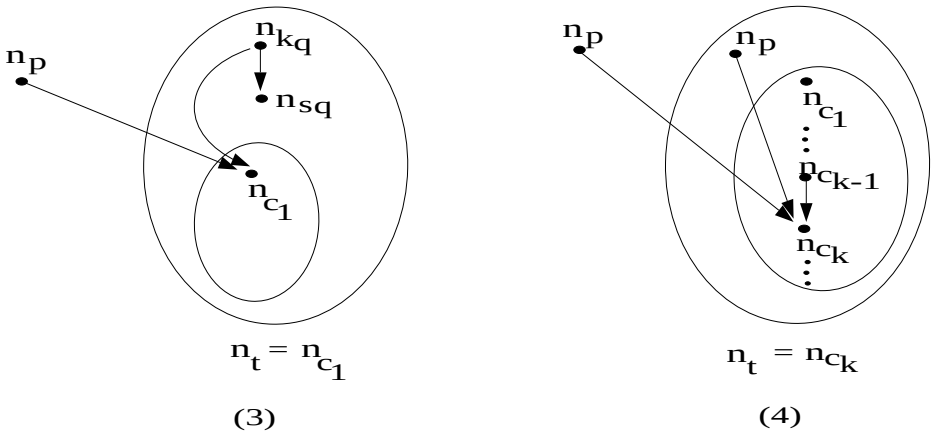


Figure 3.5: Case 3 and 4 of **construct** procedure.

3. n_t **is the first task** n_{c_1} **in** LC.

Refer to Figure 3.5(3). We check the termination condition of **Extract** to determine the start time of n_{c_1}. For case 1 of extraction, n_{c_1} has no predecessor in NLC; thus moving it into processor j will not affect its ready time since the processor architecture

is a clique. For case 3 of extraction, $pred_l(n_{c_1}) = (n_{k_1}, ..., n_{k_q})$ and there exists a task n_{s_q} in NLC satisfying $n_{k_q} \preceq_i n_{s_q} \preceq_i n_{c_1}$ in S_l. Therefore,

$$ST_l(n_{c_1}) \geq \max\{P_l(n_{c_1}, pred_l(n_{c_1})),\ Q_l(n_{c_1}, R_l(n_{c_1})),\ ST_l(n_{s_q}) + \tau_{s_q}\}$$

$$RT_{l+1}(n_{c_1}) = Q_{l+1}(n_{c_1}, PRED(n_{c_1})) = \max\{Q_{l+1}(n_{c_1}, R_l(n_{c_1})), Q_{l+1}(n_{c_1}, pred_l(n_{c_1}))\}.$$

We now show $ST_l(n_{s_q}) + \tau_{s_q} \geq Q_{l+1}(n_{c_1}, pred_l(n_{c_1}))$.

Since $n_{k_q} \in pred_l(n_{c_1})$ and it is executed after all other tasks in $pred_l(n_{c_1})$,

$$ST_l(n_{k_q}) + \tau_{k_q} + c_{k_q,c_1} \geq Q_{l+1}(n_{c_1}, pred_l(n_{c_1})).$$

Because of $n_{k_q} \preceq_i n_{s_q}$ and $\tau_{s_q} \geq c_{k_q,c_1}$,

$$ST_l(n_{s_q}) + \tau_{s_q} \geq ST_l(n_{k_q}) + \tau_{k_q} + c_{k_q,c_1}.$$

Then we have the following

$$ST_l(n_{s_q}) + \tau_{s_q} \geq Q_{l+1}(n_{c_1}, pred_l(n_{c_1})).$$

Also we know

$$Q_{l+1}(n_{c_1}, R_l(n_{c_1})) \leq Q_l(n_{c_1}, R_l(n_{c_1})).$$

Consequently,

$$RT_{l+1}(n_{c_1}) \leq ST_l(n_{c_1}).$$

Since processor j contains a linear cluster, we can set

$$ST_{l+1}(n_{c_1}) = RT_{l+1}(n_{c_1}) \quad \Longrightarrow \quad ST_{l+1}(n_{c_1}) \leq ST_l(n_{c_1}).$$

4. **n_t is the k-th node n_{c_k} in LC**

Refer to Figure 3.5(4). We examine the relationship between $n_{c_{k-1}}$ and n_{c_k} in case 2 of the extraction procedure. Since $pred_l(n_{c_k}) = (n_{k_1}, ..., n_{k_q})$, and $k_q = c_{k-1}$ and there is no successor $n_{s_{k-1}}$ of $n_{c_{k-1}}$ satisfying $n_{c_{k-1}} \preceq_i n_{s_{k-1}} \preceq_i n_{c_k}$ in S_l,

$$\begin{aligned} ST_l(n_{c_k}) &\geq \max\{P_l(n_{c_k}, pred_l(n_{c_k})),\ Q_l(n_{c_k}, R_l(n_{c_k}))\} \\ &\geq \max\{ST_l(n_{c_{k-1}}) + \tau_{c_{k-1}},\ Q_l(n_{c_k}, R_l(n_{c_k}))\} \end{aligned}$$

$$RT_{l+1}(n_{c_k}) = \max\{ST_{l+1}(n_{c_{k-1}}) + \tau_{c_{k-1}}, Q_{l+1}(n_{c_k}, PRED(n_{c_k}) - \{n_{c_{k-1}}\})\}$$

$$Q_{l+1}(n_{c_k}, PRED(n_{c_k}) - \{n_{c_{k-1}}\})$$
$$= \max\{Q_{l+1}(n_{c_k}, R_l(n_{c_k})), Q_{l+1}(n_{c_k}, pred_l(n_{c_k}) - \{n_{c_{k-1}}\})\}.$$

Thus from the induction assumption we have that

$$Q_{l+1}(n_{c_k}, R_l(n_{c_k})) \leq Q_l(n_{c_k}, R_l(n_{c_k})).$$

Since the graph is coarse then $\tau_{k_q} \geq \max_{1 \leq b \leq q-1}\{c_{k_b,c_k}\}$ and since $n_{k_1} \preceq_i, ..., \preceq_i n_{k_q}$, we have that

$$\begin{aligned} ST_l(n_{c_{k-1}}) + \tau_{c_{k-1}} &\geq P_{l+1}(n_{c_k}, pred_l(n_{c_k}) - \{n_{c_{k-1}}\}) + \max_{1 \leq b \leq q-1}\{c_{k_b,c_k}\} \\ &\geq Q_{l+1}(n_{c_k}, pred_l(n_{c_k}) - \{n_{c_{k-1}}\}). \end{aligned}$$

Therefore, we have shown that

$$RT_{l+1}(n_{c_k}) \leq ST_l(n_{c_k})$$

and since processor j contains a linear cluster, we can set

$$ST_{l+1}(n_{c_1}) = RT_{l+1}(n_{c_1}), \quad \text{and thus } ST_{l+1}(n_{c_1}) \leq ST_l(n_{c_1})$$

□

Example: We explain how the above procedure works for Figure 3.6.

Assume that the following nonlinear clustering of the DAG in Figure 3.6 is given,

$$CLUSTER(0) = \{M_1 = \{n_7\}, \ M_2 = \{n_6\}, \ M_3 = \{n_5\}, \ M_4 = \{n_1, n_2, n_3, n_4\}\}.$$

The optimum schedule S_0 with parallel time $PT_0 = 11$ is shown in Figure 3.6(b). The NLC cluster is M_4. The node n_4 is extracted first and $pred_0(n_4) = (n_2, n_3)$ and $LC = \{n_4\}$. Because of case 2, the node n_3 is extracted next and $pred_0(n_3) = (n_1)$ and $LC = \{n_3, n_4\}$. Since $n_1 \preceq_i n_2 \preceq_i n_3$ the extraction stops because of case 3. We rename LC to be M_5, and the linear clustering is,

$$CLUSTER(1) = \{M_1 = \{n_7\}, M_2 = \{n_6\}, M_3 = \{n_5\}, M_4 = \{n_1, n_2\}, M_5 = \{n_3, n_4\}\}.$$

In Figure 3.6(b) S_0 is the schedule at the beginning of the first step of the linearization procedure. At the end of that step the linear cluster M_5 is extracted. A schedule

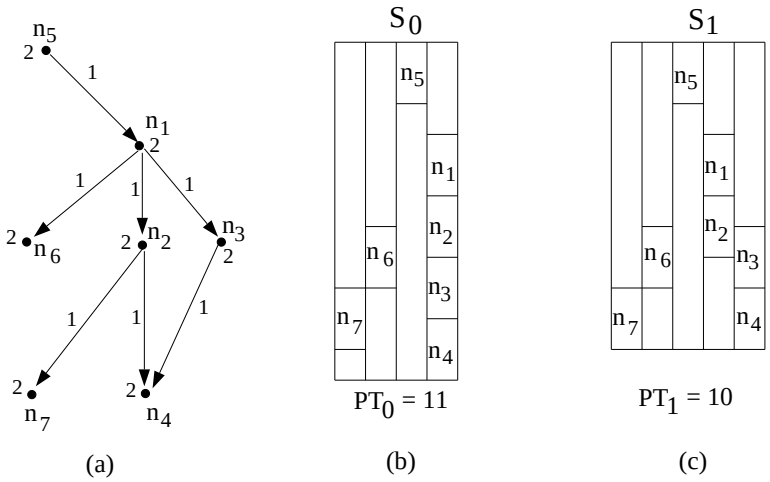


Figure 3.6: An example of linearization with $c_{i,j} = 1$ and $\tau_i = 2$, $i = 1 : 7$. S_0 is the optimum scheduling of a nonlinear clustering. S_1 is a linear clustering with less parallel time.

S_1 is constructed by following **Construct** procedure and shown in Figure 3.6(c). This schedule satisfies $ST_l(n_t) \geq ST_{l+1}(n_t)$ for all nodes n_t. Notice also that the execution order of S_1 is the same as S_0, i.e. $n_3 \preceq_i n_4$ in S_0 and S_1, and that the parallel time of S_1 is less than the parallel time of S_0.

This theorem shows that the solution for a coarse grain DAG is equivalent to that of finding an optimal linear clustering. Picouleau [79] has shown that the scheduling problem for coarse grain DAGs is NP-complete, therefore optimal linear clustering is NP-complete.

Corollary 3.1 *Optimal linear clustering is NP-complete.*

Even though linear clustering is still very difficult, for coarse grain DAGs any linear clustering algorithm guarantees performance within a factor of two of the optimum. This bound is the special case of the following theorem described in Gerasoulis and Yang [42] and the proof is based on Gerasoulis and Venugopal [41].

Theorem 3.2 *Let PT_{opt} be the optimum parallel time and PT_{lc} be the parallel time of a linear clustering, then $PT_{lc} \leq (1 + \frac{1}{g(G)})PT_{opt}$.*

3.3 Experiments on Granularity Issues

The granularity of a program task graph is determined by program partitioning. Our analysis in the previous section shows that a program partitioning with $g(G) \geq 1$ is suitable to fully utilize processor resources. If a partitioning results in a fine grain DAG, a scheduling algorithm may have to sequentialize fine grain parts of a task graph. Since optimal nonlinear scheduling is usually expensive in terms of complexity cost, starting from a coarser grain partitioning with sufficient parallelism is a better approach. Further more, coarse grain partitioning could increase the size of messages and as a result it saves the effort in message vectorization [1].

As demonstrated in Section 3.1.1, if a DAG is produced using a statement-level partitioning (i.e. each task performs a simple arithmetic operation and communicates with others a scalar variable), then in general the granularity value of such a DAG is very small on message-passing architectures because of large start-up overhead in inter-processor communication. Exploiting fine grain statement-level parallelism will result in poor performance. To increase the granularity of a program DAG, we should use coarse grain procedure-level or loop-level partitioning.

We consider the example of kji Gauss-Jordan (GJ) algorithm with no pivoting [74].

for $k = 1 : n$
  for $j = k + 1 : n + 1$
    $a_{k,j} = a_{k,j}/a_{k,k}$
    for $i = 1 : n$ **and** $i \neq k$
      $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}$
    endfor
  endfor
endfor

3.3.1 Statement level fine grain parallelism

If we take each assignment statement as a task, we get a three dimensional dependence graph with the degree of parallelism equal to n^2, S.Y. Kung [63], pp. 170. Each

[1] A program optimization used in ID Nouveau [86] is message vectorization which combines smaller messages into a larger message.

task performs a simple arithmetic operation and communicates with each other in one floating point number. An estimation of the granularity for this DAG is $g(G) \approx \omega/(\alpha+\beta)$ where ω is the time for one addition or subtraction, plus one multiplication. In Table 3.1, we list the parameters of several architectures for single-precision arithmetic and the value of $\omega/(\alpha+\beta)$, see [32]. We observe that the granularity value is too small for exploiting parallelism on these architectures.

	iPSC/2	iPSC/860	nCUBE-I	nCUBE-II
α (microseconds)	697	136	383.6	200
β (microseconds/word)	1.6	1.6	10.4	2.4
ω (microseconds/flop)	11.4	0.2	35.1	1.6
$g(G) = \omega/(\alpha+\beta)$	0.016	0.0015	0.089	0.0079

Table 3.1: Architecture parameters for single-precision arithmetic.

3.3.2 Interior loop partitioning

To increase the granularity, we can use interior loop partitioning as shown in Figure 3.7. The task T_k^j defined in Figure 3.7 uses column k to modify column j of the matrix A. The matrix is partitioned into columns consistent with the data accessing pattern of tasks. The new dependence graph is given in Figure 3.8 [26, 69]. Task T_k^{k+1} is a broadcasting node, sending the same column $k+1$ to all T_{k+1}^j, $j = k+2 : n+1$. This new DAG has degree of parallelism equal to n.

Gauss-Jordan kji form

for $k = 1 : n$
  for $j = k+1 : n+1$
T_k^j : { $a_{k,j} = a_{k,j}/a_{k,k}$
    for $i = 1 : n$ **and** $i \neq k$
      $a_{i,j} = a_{i,j} - a_{i,k} * a_{k,j}$
    end}
  end
end

Figure 3.7: The interior loop task partitioning for GJ.

The communication and computation weights are estimated as follows. For T_k^j there are about $n\omega$ operations for each task, where ω is the time for performing "$a_{i,j} =$

$a_{i,j} - a_{i,k} * a_{k,j}$". The communication cost is $\alpha + n\beta$ which is time to transfer a column between two neighbor processors. The granularity of the GJ DAG in Figure 3.8 is $g(GJ) = \frac{n\omega}{\alpha+n\beta}$. For a sufficiently large n, $g(GJ) \approx \omega/\beta$.

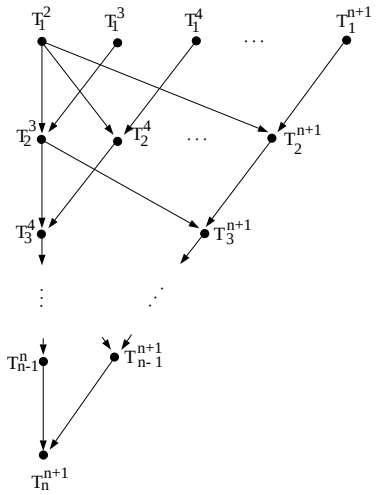


Figure 3.8: The GJ DAG. The node computation cost is $\tau = n\omega$ and edge communication cost is $c = \alpha + n\beta$.

In the literature, the following *natural clustering* has been used for this GJ DAG which assumes that each cluster contains all tasks that modify the same column [26, 39, 77, 85, 87] [2].

$$M_j = \{T_k^j \mid k = 1 : j-1\}, \quad j = 2 : n+1$$

In [42], we prove the following result:

Theorem 3.3 *If $g(GJ) \geq 1$, the natural linear clustering is optimal for both clique and ring architectures.*

Table 3.2 gives the granularity values of the GJ DAG for different architectures. For the nCUBE-I, if $n \geq 16$ then the GJ graph is coarse grain implying that the optimum parallel time will be derived by the natural linear clustering. The same is true for iPSC/2 but in this case the size of the problem must be greater than 72. However, for the newest generation of the message passing architectures, the iPSC/860 and nCUBE-II, the GJ DAG based on the interior loop partitioning is fine grain for any size of

[2] This clustering can also be derived by the owner computes rule

the matrix. Therefore according to our granularity theory presented in the previous sections, we cannot guarantee the performance of linear clustering. Thus nonlinear clustering may be needed.

	iPSC/2	iPSC/860	nCUBE-I	nCUBE-II
$g(GJ) \geq 1$ when $n \geq$	72		16	
$\lim_{n\to+\infty} g(GJ) = \omega/\beta =$	7.1	0.125	3.4	0.667
$\lim_{n\to+\infty} PT_{lc}/PT_{opt} \leq 1+\beta/\omega =$	1.14	9.0	1.29	2.5

Table 3.2: Granularity values of the GJ DAG on different architectures.

One nonlinear clustering for the GJ DAG is to wrap n linear clusters around a ring of p processors with $p < n$ [26, 39, 77]. Table 3.3 lists the result of linear clustering when $n = 128$ and $p = 128$ on nCUBE-II and also the results of wrap nonlinear clustering when $p = 8, 16, 64$. PT is the parallel time in milliseconds. We see that as p decreases, wrap nonlinear clustering performs well up to the point when $p = 32$. Afterwards the performance becomes worse because parallelism is reduced too much, compared to the gain from the elimination of communication cost.

3.3.3 Block partitioning

We could use other program partitionings to increase the granularity of the GJ algorithm. A popular approach is to use a submatrix data partition known as the BLAS-3 block partition [31]. The matrix of $n \times n$ is divided into $N \times N$ submatrices and each submatrix has size of $r \times r$ where $N = n/r$. Each task T_k^j in the block GJ DAG is operating on a block of columns composed of N submatrices. The task definition is given below:

p	128	64	32	16	8
PT	855.3	847.8	630.1	780.3	1202.9

Table 3.3: The parallel time of linear and nonlinear clustering for $n = 128$ on nCUBE-II.

$$
\begin{array}{l}
T_k^j : \{ \quad A_{k,j} = A_{k,k}^{-1} * A_{k,j} \\
\qquad \textbf{for } i = 1 : N \textbf{ and } \ i \neq k \\
\qquad\quad A_{i,j} = A_{i,j} - A_{i,k} * A_{k,j} \\
\qquad \textbf{end}\}
\end{array}
$$

The dependence structure of the GJ DAG remains the same as in Figure 3.8 but the degree of parallelism is reduced from n to $N = n/r$. Also the computation size of T_k^j increases to about $Nr^3\omega$ and a message communicated between tasks is a block column of size Nr^2. Thus the granularity $g(GJ) \asymp \frac{r\omega}{\beta}$ for a sufficiently large matrix, has increased by a factor of r. To make $g(GJ) \geq 1$ the minimum submatrix size r is $\lceil \frac{\beta}{\omega} \rceil$ which is 2 for nCUBE-II and 8 for i860. Namely, one must use a partitioning on the iPSC/860 about 4 times coarser than that used on nCUBE-II.

Figure 3.9 gives the experimental results on nCUBE-II with 128 processors using the block GJ algorithm with $n = 128$. It lists the parallel time in milliseconds for two types of broadcasting algorithms [87]: (a) the gray-code ring broadcasting (b) the hypercube spanning tree. We vary the granularity by varying the size of the submatrix partitioning r. The parallel time for ring broadcasting is slightly smaller than that of tree broadcasting when $\log p \leq 6$, but is much smaller for larger $\log p = 7$. This result is consistent with Saad [87].

When $r = 1$, GJ has the worst performance for $p = 128$ processors because the granularity is too fine and the parallelism implemented by linear clustering is not useful. When $r \geq 2$, the granularity increases, the GJ DAG becomes coarse grain and linear clustering performs well. Because increasing the granularity has the adverse effect of reducing the parallelism the performance of linear clustering will degrade beyond a certain granularity value as it is the case in Figure 3.9 when $r > 4$. Notice the dramatic improvement in performance when when we go from a fine grain DAG ($r = 1$) to a coarse grain DAG ($r = 2$). This example shows that our granularity definition captures a point when parallelism becomes useful.

Finally, we verify the result shown in Theorem 3.1 that linear clustering outperforms nonlinear clustering for a coarse grain DAG. The theoretical estimation indicates that for $r \geq 2$, BLAS-3 GJ DAG is coarse grain. We will compare the natural linear

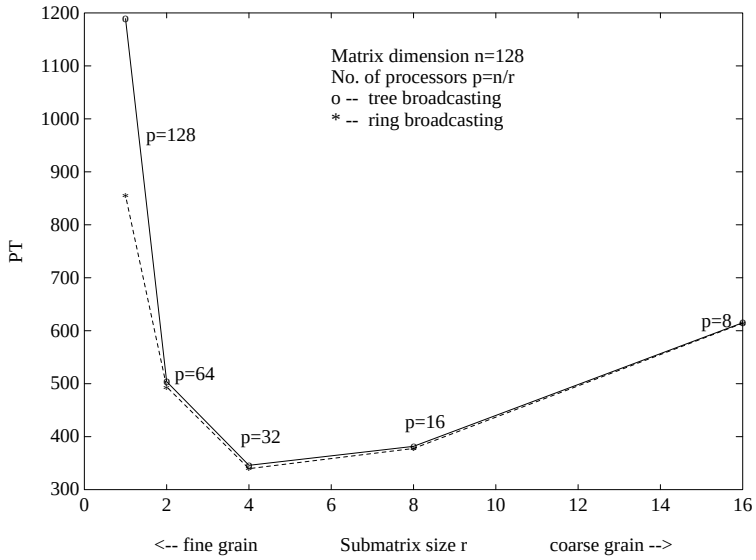


Figure 3.9: Performance of linear clustering for block GJ DAGs on nCUBE-II. When $r = 1$, the DAG is column partitioning and is fine grain. When $r \geq 2$, it is coarse grain. PT is the parallel time in milliseconds.

clustering with a nonlinear clustering that wraps N linear clusters into p nonlinear clusters. In Figure 3.10, we show the parallel time for block size $r = 2$ and $r = 4$ on the nCUBE-II. We also mark the number of linear or nonlinear clusters (#NLC and #LC) in each case. When $r = 4$, the natural linear clustering always has shorter parallel time. When $r = 2$, the time for the natural clustering (#LC=64) is slightly longer than the nonlinear clustering with #NLC=32 but is shorter than that with #NLC=16 and 8. For this case, some extra communication overhead (e.g. message buffering and physical distance) also exist, which slightly affects performance.

3.4 Concluding Remarks

In this chapter, we have considered the question of sequentialization vs. parallelization of task graphs. We have shown that the granularity of a task graph affects the optimum trade off point and if the granularity is greater than one then linear clustering is optimum. Linear clustering preserves the parallelism and is suitable for coarse grain tasks and nonlinear clustering reduces the parallelism in order to save communication cost. Our analysis is based on the global quantity of granularity defined as a minimum of

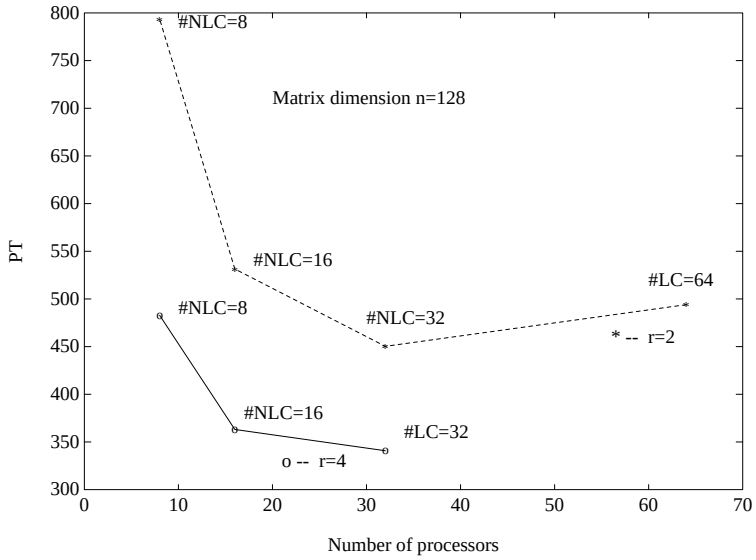


Figure 3.10: Linear and nonlinear clustering on BLAS-3 GJ DAG on nCUBE-II with different processors. PT is the parallel time in millisecond.

the task grains over all tasks in a graph. The theorems of this chapter could be used in cases where the granularity assumptions are partially true. However in general, for fine grain parts of a graph, nonlinear clustering is necessary and additional investigation is needed to bound the performance for such clustering.

To conclude, when clustering a DAG a heuristic algorithm should select an appropriate strategy based on the granularity information and its impact on the parallel time. We will discuss such an algorithm in the next chapter.

Chapter 4

Task Graph Clustering

In this chapter, we consider *clustering tasks* by scheduling tasks on an unlimited number of fully-connected processors. Two tasks in the same cluster are scheduled in the same processor in the final parallel code.

The granularity analysis in the previous section shows that the basic principle of a clustering algorithm is to select clustering strategies according to granularity value: sequentialize fine grain tasks to avoid high communication cost using nonlinear clustering and preserve coarse grain task parallelism using linear clustering. However, how to incorporate such strategies to obtain the optimal or near optimal solution is a challenging problem.

4.1 Complexity Issue and Previous Approaches

The complexity of this clustering problem has been found much more difficult than that of the classical scheduling problem where the communication cost is ignored, Graham [48] and Lenstra and Kan [66]. The NP-completeness of clustering for parallel time minimization has been shown by Sarkar [90], Chretienne [20] and Papadimitriou and Yannakakis [78]. For clustering special classes of DAGs, Chretienne [21] shows that the problems of scheduling a join, fork DAG or coarse grain tree DAG are solvable in a polynomial time, but the complexity jumps to NP-complete for scheduling fine-grain tree DAGs and a DAG structure by concatenating a fork and a join together.

Basically there have been two approaches in the literature addressing this scheduling problem. The first approach considers heuristics for arbitrary DAGs and the second studies optimal algorithms for special classes of DAGs. When task duplication is allowed, Papadimitriou and Yannakakis [78] have proposed an approximate algorithm for

a DAG with equal task weights and equal edge weights, which guarantees a performance within 50% of the optimum. This algorithm has a complexity of $O(v^3(v \log v + e))$ where v is the number of tasks and e is the number of edges. Kruatrachue and Lewis [62] have also given an $O(v^4)$ algorithm for a general DAG based on task duplication. One difficulty with allowing task duplication is that it increases the space complexity when executing parallel programs on a real machine and as a result the algorithms might not be practical.

Without task duplication, many heuristic scheduling algorithms for arbitrary DAGs have been proposed in the literature, e.g. Kim and Browne [60], Sarkar [90], Wu and Gajski [98], Girkar and Polychronopoulos [46]. One difficulty with most existing algorithms for general DAGs is their high complexity. Another is that none of these general algorithms determines the optimum for special DAGs such as join, fork and coarse grain tree. Even though polynomial time algorithms for special types of DAGs exist in the literature, Chretienne [21], Anger, Hwang and Chow [6], these algorithms are not general. *As far as we know no scheduling algorithm exists that works well for arbitrary graphs, finds optimal schedules for special DAGs and also has a low complexity.* We will present one such algorithm with a complexity of $O((v + e) \log v)$, called the Dominant Sequence Clustering (DSC) algorithm.

The organization is as follows: Section 4.2 introduces the basic concepts and a framework for the design of clustering algorithms. Section 4.3 describes an initial design of the DSC algorithm and analyzes its weaknesses. Section 4.4 presents an improved version of DSC that takes care of the initial weaknesses and analyzes how DSC achieves both low complexity and good performance. Section 4.5 gives a performance bound for a general DAG. It shows that the performance of DSC is within 50% of the optimum for coarse grain DAGs, and it is optimal for join, fork, coarse grain trees and a class of fine grain trees. Section 4.6 presents experimental results and compares the performance of several algorithms from the literature with DSC.

4.2 Preliminaries

In Chapter 2, we have described the Gantt chart representation of a schedule solution for a DAG. The solution of scheduling can be considered as two parts: (1) processor assignment, also called clustering in this chapter and (2) task ordering for each processor. Thus we can represent a schedule using a *scheduled* DAG. A scheduled DAG is composed of both clustering and task execution ordering information. Figure 4.1(b) is a Gantt chart representation of a scheduling solution. It can also be represented graphically in Figure 4.1(c), which is called a *scheduled DAG*. For example, the dashed edge between n_3 and n_4 in Figure 4.1(c) represents the execution order imposed by such a schedule. $CLUST(n_x)$ stands for the cluster of node n_x. If a cluster contains only one task, it is called a *unit* cluster.

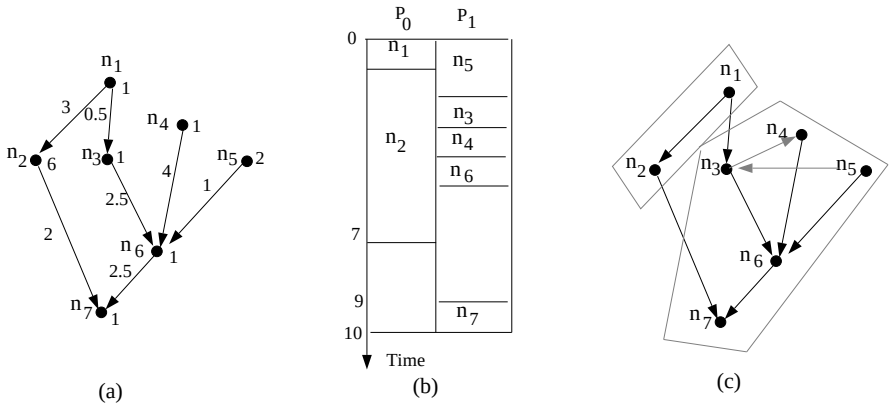


Figure 4.1: Scheduling example: (a) is a DAG. (b) is a Gantt graph representation of schedule. (c) is a scheduled DAG.

A DAG with a given clustering but without task execution order imposed is called a *clustered graph*. A communication edge weight in a clustered graph becomes zero if the start and end nodes of this edge are in the same cluster. The *critical path* of a clustered graph is the longest path in that graph including nonzero communication cost. The parallel time of executing a clustered DAG is not determined by the critical path of this DAG but by the critical path of the scheduled DAG where the nonlinear cluster is linearized by counting zero-weighted pseudo edges. We call the critical path of a scheduled DAG as *dominant sequence*, DS for short. For Figure 4.1(c), the critical path of the clustered graph is $< n_1, n_2, n_7 >$. The DS is still that path. If the weight

of n_5 were changed from 2 to 6, the critical path of clustered graph remains the same $< n_1, n_2, n_7 >$, but the DS would be the task sequence $< n_5, n_3, n_4, n_6, n_7 >$.

Let $tlevel(n_x)$ be the length of the longest path including the zero-weighted pseudo edges from an entry node to n_x in the scheduled DAG (excluding the weight of n_x). Symmetrically, let $blevel(n_x)$ be the length of the longest path from n_x to an exit node (including the weight of n_x). For example, in Figure 4.1(c), $tlevel(n_1) = 0$,$blevel(n_1) = 10$, $tlevel(n_3) = 2, blevel(n_3) = 4$. The following formula can be used to determine the parallel time from a scheduled graph:

$$PT = max\{tlevel(n_x) + blevel(n_x) | n_x \in V\}. \tag{4.1}$$

4.2.1 Scheduling as successive clustering refinements

Our approach for solving the scheduling problem with unlimited resources is to consider the scheduling algorithms as performing a sequence of clustering refinement steps. As a matter of fact most of the existing algorithms can be characterized by using such a framework [44]. The initial step assumes that each node is mapped in a unit cluster. In other steps the algorithm tries to improve the previous clustering by merging appropriate clusters. A merging operation is performed by zeroing an edge cost connecting two clusters [1].

Sarkar's algorithm

We consider Sarkar's algorithm [90], pp. 123-131, as an example of an edge-zeroing clustering refinement algorithm. This algorithm first sorts the e edges of the DAG in a decreasing order of edge weights and then performs e clustering steps by examining edges from left to right in the sorted list. At each step, it examines one edge and zeros this edge if the parallel time does not increase.

Sarkar's algorithm requires the computation of the parallel time at each step and this problem is NP-hard in stong sense [52]. Sarkar uses the following strategy: Order

[1] In this chapter, two clusters will not be merged if there is no edge connecting them since this cannot decrease the parallel time.

independent tasks in a cluster by using the highest *blevel* first priority heuristic, where *blevel* is the value computed in the previous step. The new parallel time is then computed by traversing the scheduled DAG in $O(v + e)$ time. Since there are e steps the overall complexity is $O(e(e + v))$.

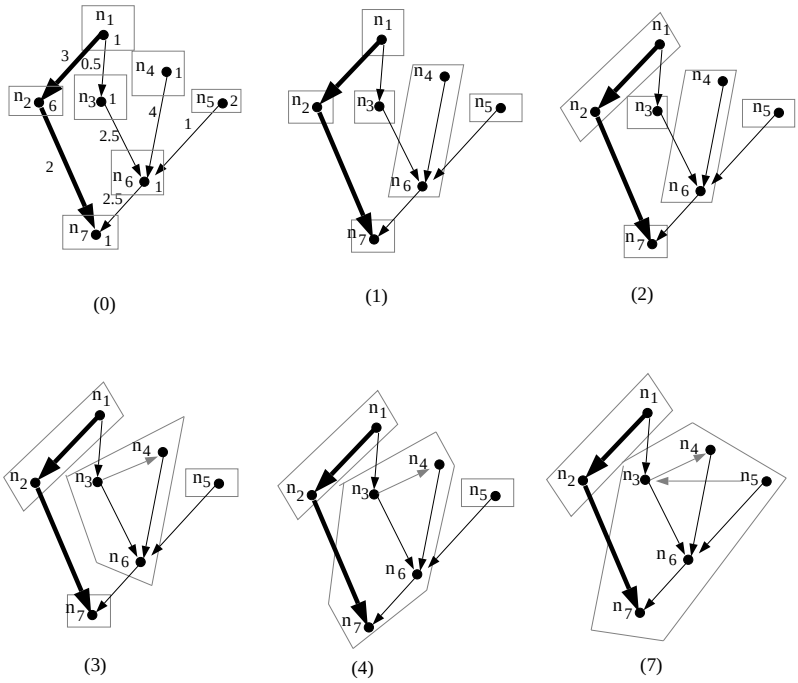


Figure 4.2: Clustering steps by Sarkar's algorithm for Figure 1(a).

Figure 4.2 shows the clustering steps of Sarkar's algorithm for the DAG in Figure 4.1(a). The sorted edge list with respect to edge weights is

$$\{(n_4, n_6), (n_1, n_2), (n_3, n_6), (n_6, n_7), (n_2, n_7), (n_1, n_3), (n_5, n_6)\}.$$

Table 4.1 traces the execution of this algorithm where PT stands for the parallel time and PT_i is the parallel time for executing the clustered graph at the completion of step i. Initially each task is in a separate cluster as shown in Figure 4.2(0) and the thick path indicates the DS whose length is $PT_0 = 13$. At step 1, edge (n_4, n_6) is examined and PT remains 13 if this edge is zeroed. Thus this zeroing is accepted. In step 2, 3 and 4, shown in Figure 4.2(2), (3) and (4), all examined edges are zeroed since each zeroing does not increase PT. At step 5, edge (n_2, n_7) is examined and by zeroing it the parallel time increases from 10 to 11. Thus this zeroing is rejected. Similarly, at step 6 zeroing (n_1, n_3) is rejected. At step 7 (n_5, n_6) is zeroed, a pseudo-edge from n_5 to n_3 is

added because after step 6 $blevel(n_3) = 3$ and $blevel(n_5) = 5$. Finally two clusters are produced with PT=10.

step i	edge examined	PT if zeroed	zeroing	PT_i
0				13
1	(n_4, n_6)	13	yes	13
2	(n_1, n_2)	10	yes	10
3	(n_3, n_6)	10	yes	10
4	(n_6, n_7)	10	yes	10
5	(n_2, n_7)	11	no	10
6	(n_1, n_3)	11	no	10
7	(n_5, n_6)	10	yes	10

Table 4.1: Clustering steps of Sarkar's algorithm corresponding to Figure 4.2.

4.3 An Initial Design of the DSC Algorithm

4.3.1 Design considerations for the DSC algorithm

As we saw in the previous section, Sarkar's algorithm zeroes the highest communication edge. This edge, however, might not be in a DS that determines the parallel time and as a result the parallel time might not be reduced at all, see the zeroing of edge (n_4, n_6) in step 1 Figure 4.2. In order to reduce the parallel time, we must examine the schedule of a clustered graph to identify a DS and then try to reduce its length. *The main idea behind the DSC algorithm is to perform a sequence of edge zeroing steps with the goal of reducing a DS at each step.* The challenge is to implement this idea so that the complexity of the algorithm is low so that the algorithm is able to handle large graphs. Thus the DSC algorithm has three goals:

- **G1:** The complexity should be low.
- **G2:** The parallel time should be minimized.
- **G3:** The efficiency should be maximized, by reducing the number of unnecessary clusters.

There are several difficulties in the implementation of algorithms that satisfy the above goals which we discuss below:

1. The goals G1, G2 and G3 could conflict with each other. For example the maximization of the efficiency conflicts with the minimization in the parallel time. When such conflicts arise, then G1 prevails over G2 and G3, and G2 prevails over G3.

2. Let us assume that the DS has been determined. Then there is a decision to be made for selecting edges to be examined[2] and zeroed. Consider the example in Figure 4.2(0). Initially the DS is $< n_1, n_2, n_7 >$. To reduce the length of that DS, we need to zero at least one edge in DS. Hence we need to decide which edges should be zeroed. We could zero either one or both edges. If the edge (n_1, n_2) is zeroed, then the parallel time reduces from 13 to 10. If (n_2, n_7) is zeroed the parallel time reduces to 11. If both edges are zeroed the parallel time reduces to 9.5. Therefore there are many possible ways of edge zeroings and we discuss three approaches:

   - **AP1: Multiple DS edge zeroing with maximum PT reduction.**

     This is a greedy approach that will try to get the maximum reduction of the parallel time at each clustering step.

   - **AP2: One DS zeroing of maximum weight edge.**

     Considering a DS could become a SubDS when only one edge of this DS is zeroed, we do not need to zero many edges at one step. Thus another approach is to zero one edge at one step, say the largest weight edge.

   - **AP3: One DS edge zeroing whenever it is possible with low complexity.**

     Determining a DS for a clustered DAG could take at least $O(v+e)$ time if the computation is not done incrementally. Repeating this computation for all steps will result in at least $O(v^2)$ complexity. Thus we need an incremental computation of DS from one step to the next to avoid the traversal of the entire DAG at each step. Instead of zeroing the highest edge weight we

[2]Backtracking is not used to avoid high complexity.

zero one DS edge as long as we can compute the next DS incrementally. If we cannot zero a DS edge to reduce the parallel time at some step, say because the complexity will increase, then we can zero non-DS edges and try to reduce the number of unnecessary clusters to satisfy goal G3. Also those non-DS edges could become DS in future steps and thus zeroing them may benefit the reduction of parallel time.

It is not clear how to implement AP1 or AP2 with a low complexity and also there is no guarantee that AP1 or AP2 will be better than AP3. We will use AP3 to develop our algorithm.

3. Since we do not allow backtracking the only zeroings steps that we should allow are the ones that do not increase the parallel time from one step to the next:

$$PT_{i-1} \geq PT_i.$$

Sarkar imposes this constraint explicitly in his edge zeroing process, by comparing the parallel time at each step. Here we will use an implicit constraint to avoid the explicit computation of parallel time in order to reduce the complexity.

In the next subsection, we present an initial version of DSC algorithm and then identify its weaknesses so that we can improve its performance.

4.3.2 DSC-I: An initial version of DSC

The key steps in the DSC algorithm are described below:

1. The DSC algorithm needs to identify the DS at each step so that zeroing decisions can be made. Equation (4.1) is used for this purpose. We notice that a node n_f is in DS if $tlevel(n_f) + blevel(n_f)$ is the highest value among the graph nodes. Thus if the priority for each task at each step is

$$PRIO(n_f) = tlevel(n_f) + blevel(n_f)$$

then we can identify all DS nodes as the ones with the highest priority. This implies that *tlevel* and *blevel* need to be computed at each step of the algorithm

for every node, which could be too costly in terms of complexity. Because not every DS edge is considered for zeroing at each step, we only need to determine the priority values for a small group of tasks. We also need to find a good edge zeroing traversal which makes easy the re-computation of priority values at the next step.

2. The DSC algorithm uses a *topological traversal* of the graph. During the execution of the algorithm the graph consists of two parts, the examined graph EG and the unexamined graph UEG. At each step, a node from UEG is deleted and added to EG. Initially all nodes are marked *unexamined*. The algorithm selects only an unexamined *free*[3] node with the highest priority and examines its incoming edge for zeroing or not. If there is a free node in a DS then the algorithm will try to zero its incoming edge first, otherwise it tries to zero the incoming edge of the free node of the next SubDS and so on.

3. The criterion for accepting a zeroing or not is the minimization of $tlevel(n_f)$, where n_f is the free node with the highest priority. If $tlevel(n_f)$ does not increase from the previous step by zeroing its incoming edges then such zeroings are accepted otherwise no zeroing is accepted. Notice that by reducing $tlevel(n_f)$ all paths going through n_f could be compressed and as a result the DS length could be reduced.

An algorithmic description of DSC-I is given in Figure 4.3. DSC-I satisfies the following properties:

Equation (4.1) implies that reducing the priority value of tasks would lead to the reduction of the parallel time. Thus the constraint that *tlevel* values do not increase implies:

Property 4.1 $PT_{i-1} \geq PT_i$.

A formal proof will be given in Section 4.4.5.

[3]A node is called free if all of its predecessors have been examined.

$EG = \emptyset$. $UEG = V$.
Compute *blevel* for each node and set $tlevel = 0$ for each entry node.
Every task is marked *unexamined* and assumed to constitute one unit cluster.
While there is a node unexamined **Do**
    Find a free node n_f with highest priority from UEG.
    Merge[a] n_f with the cluster of one of its predecessors such that $tlevel(n_f)$ decreases in a maximum degree. If all zeroings increase $tlevel(n_f)$, n_f remains in a unit cluster.
    Update the priority values of n_f's successors.
    $UEG = UEG - \{n_f\}$; $EG = EG + \{n_f\}$.
EndWhile

[a]When a free task n_f is merged to the cluster where one of its predecessors resides, a pseudo-edge is added from the last task of this cluster to n_f if they are independent.

Figure 4.3: The initial design of the DSC algorithm.

If *tlevel* and *blevel* information is re-used after each step, then the complexity in determining DS nodes could be reduced. The following property explains how the topological traversal and the cluster merging rule defined in the DSC-I description make the complexity reduction possible.

Property 4.2 *For the DSC-I algorithm, $tlevel(n_x)$ remains constant if $n_x \in EG$ and $blevel(n_x)$ remains constant if $n_x \in UEG$.*

Proof: If $n_x \in UEG$, then the topological traversal implies that all descendant of n_x are in UEG. Since n_x and its descendants are in separate unit clusters, $blevel(n_x)$ remains unchanged before it is examined. Also for nodes in EG, all clusters in EG can be considered "linear" by counting the pseudo execution edges. When a free node is merged to a "linear" cluster it is always attached as the last node of that "linear" cluster. Thus $tlevel(n_x)$ remains unchanged after n_x has been examined. □

Property 4.3 *The time complexity of DSC-I algorithm is $O(e + v \log v)$.*

Proof: From Property 4.2, the priority of a free node n_f can be easily determined by using

$$tlevel(n_f) = \max_{n_j \in PRED(n_f)} \{tlevel(n_j) + \tau_j + c_{j,f}\}.$$

Once $tlevel(n_j)$ is computed after its examination at some step, where n_j is the predecessor of n_f, this value is propagated to $tlevel(n_f)$. Afterwards $tlevel(n_j)$ remains unchanged and will not affect the value of $tlevel(n_f)$ anymore.

The main computational cost of DSC-I algorithm at each step is in maintaining the priority list which is $O(\log v)$ when using a balanced tree data structure. The overall complexity after including the cost for the topological traversal of the entire graph is $O(e + v \log v)$. □

In the next subsection, we study the performance of DSC-I for some DAGs and propose modifications to improve its performance.

4.3.3 Case studies for DSC-I

In the first two examples, we demonstrate the clustering steps of DSC-I by scheduling fork and join DAGs. The reason for considering such primitive structures is that a DAG is composed of a set of join and fork components. Thus, by studying the DSC-I performance on such structures we can further understand its behavior. In the third example, we discuss a problem arising when zeroing non-DS edges.

DSC-I for fork DAGs

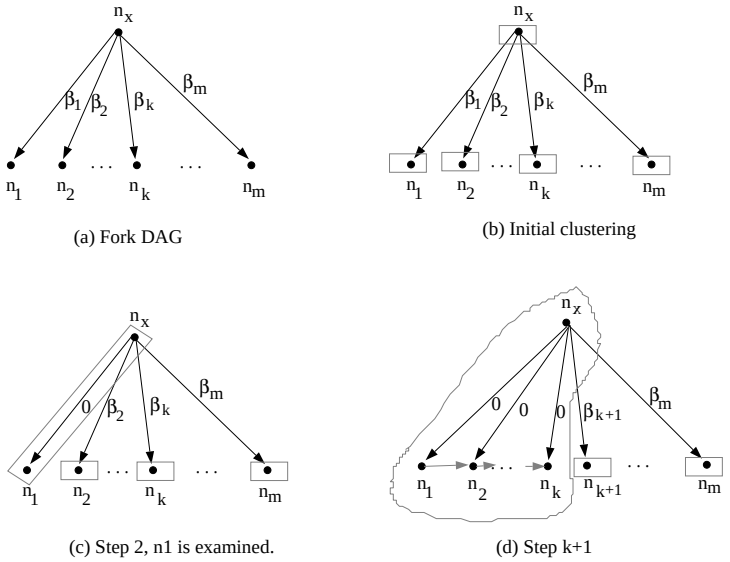


Figure 4.4: DSC-I clustering steps for a fork DAG.

Figure 4.4 demonstrates the clustering steps of DSC-I for a fork DAG. Without loss of generality, assume that the leaf nodes in this DAG shown in (a) are sorted such that $\tau_j + \beta_j \geq \tau_{j+1} + \beta_{j+1}, \quad j = 1 : m-1$. The steps of DSC-I are described below.

Figure 4.4(b) shows the initial clustering where each node is in a unit cluster. $EG = \{\}$, n_x is the only free task in UEG and $PRIO(n_x) = \tau_x + \beta_1 + \tau_1$. At step 1, n_x is selected and it has no incoming edges. It remains in a unit cluster and $EG = \{n_x\}$. After that, $n_1, n_2, \cdots, n_m$ become free and n_1 has the highest priority, $PRIO(n_1) = \tau_x + \beta_1 + \tau_1$, and $tlevel(n_1) = \tau_x + \beta_1$. At step 2 shown in (c), n_1 is selected and merged to the cluster of n_x and $tlevel(n_1)$ is reduced in a maximum degree to τ_x. At step $k+1$, n_k is selected. The original leftmost scheduled cluster in (d) is a "linear" chain $n_x, n_1, \cdots, n_{k-1}$. If attaching n_k to the end of this chain does not increase $tlevel(n_k) = \tau_x + \beta_k$, the zeroing of edge (n_x, n_k) is accepted and the new $tlevel(n_k) = \tau_x + \sum_{j=1}^{k-1} \tau_j$. Thus the condition of accepting or not a zeroing can be expressed as:

$$\sum_{j=1}^{k-1} \tau_j \leq \beta_k.$$

We now show that the DSC-I is a greedy algorithm for the fork set and AP1 and AP2 are actually satisfied for this case.

Property 4.4 *Given a fork DAG, the DSC-I always zeros a DS edge at each step.*

Proof: It suffices to show that a free node in DS is available at every step. Since this free node has the highest priority its incoming edge will be examined and if the parallel time is reducible by zeroing it will be reduced.

At step 1, in Figure 4.4 n_x is examined and $PT_1 = PRIO(n_x) = \tau_x + \beta_1 + \tau_1$. The node n_x is always in a DS. At step 2, n_1 is free and in $DS =< n_x, n_1 >$, n_1 is examined and edge (n_x, n_1) is zeroed.

Assume that before starting step $k+1$ the statement is correct. Notice that step k examines node n_{k-1}. The parallel time at the completion of step k is:

$$PT_k = \tau_x + \max(\sum_{j=1}^{k-1} \tau_j, \beta_k + \tau_k).$$

At step $k+1$, if $\sum_{j=1}^{k-1} \tau_j > \beta_k$ then the length of $DS =< n_x, n_1, \cdots, n_{k-1} >$ is not compressible. There are no unexamined DS nodes and this is true for all remaining

steps. If $\sum_{j=1}^{k-1} \tau_j \leq \beta_k$, $PT_k = PRIO(n_k)$, then n_k is free and in DS and will be examined at step $k+1$. □

As it turns out the DSC-I algorithm is an optimal algorithm for the fork and the proof will be given in section 4.5.

DSC-I for join DAGs

Let us consider the DSC-I algorithm for join DAGs.

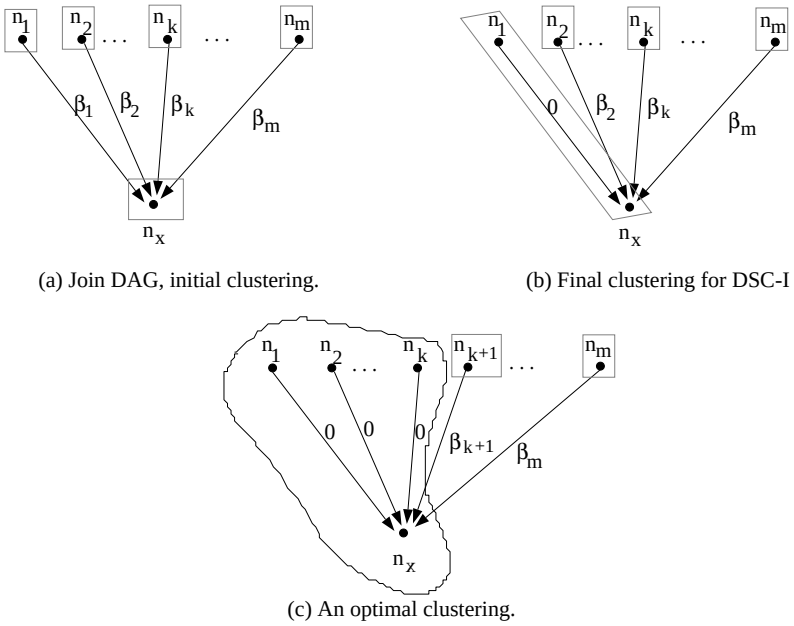


Figure 4.5: DSC-I clustering for a join DAG.

Figure 4.5 shows a join DAG, the final clustering step of DSC-I and the optimum clustering for a join DAG. Without loss of generality, assume that the entry nodes in the join DAG are sorted such that $\tau_j + \beta_j \geq \tau_{j+1} + \beta_{j+1}$, $j = 1 : m-1$. The steps of DSC-I are described below.

Figure 4.5(a) shows the initial clustering. Nodes $n_1, n_2, \cdots, n_m$ are free. One DS is $< n_1, n_x >$. Step 1 selects n_1 which is in DS. No incoming edge exists for n_1 and no zeroing is performed. The DS is still $< n_1, n_x >$ after step 1. Now n_x becomes *partially free*[4]. Step 2 selects n_2 which is not in DS. No incoming edge exists and n_2 remains

[4]A node is partial free if it is in UEG and at least one of its predecessors has been examined but not all of its predecessors have been examined.

to be in a unit cluster in EG. Step $m+1$ selects n_x which is now in the DS and then (n_1, n_x) is zeroed. The final result of DSC-I is shown in Figure 4.5(b) which may not be optimal. The optimal result for a join DAG as shown in Figure 4.5(c) is symmetric to the previous optimal solution for a fork.

The join example shows that zeroing only one incoming edge of n_x is not sufficient attain the optimum. In general, when a free node is examined, zeroing multiple incoming edges of this free node instead of zeroing one edge could result in a reduction of *tlevel* in a maximum degree. As a consequence, the length of DS or SubDS going through this node could be reduced even more substantially. To achieve such a greedy goal, a minimization procedure that zeros *multiple* incoming edges of the selected free node is needed to be introduced in the DSC algorithm.

Chretienne [20] has proposed an optimal algorithm for a fork and join, which zeroes multiple edges. The complexity of his algorithm is $O(m \log B)$ where $B = min\{\sum_{i=1}^{m} \tau_i, \beta_1 + \tau_1\} + \tau_x$. Al-Mouhamed [5] has also used the idea of zeroing multiple incoming edges of a task to compute a lower bound for scheduling a DAG, using an $O(m^2)$ algorithm, but no feasible schedules that reach the bound are produced by this algorithm. Since we are interested in lower complexity algorithms, we will use a new optimum algorithm with an $O(m \log m)$ complexity.

Dominant Sequence Length Reduction Warranty (DSRW)

We describe another problem with DSC-I. When a DS node n_y is partial free, DSC-I suspends the zeroing of its incoming edges and examines the current non-DS free nodes according to the topological order. Assume that $tlevel(n_y)$ could be reduced by δ if such a zeroing was not suspended. We should be able to get at least the same reduction for $tlevel(n_y)$ when DSC-I examines the free node n_y at some future step m. Then the length of DS going through n_y will also be reduced. However, we will see that this is not the case with DSC-I.

Figure 4.6(a) shows a weighted DAG and (b) is the result of step 2 of DSC-I after n_2 is merged to $CLUST(n_1)$. The new DS depicted with the thick arrow in (b) is $< n_1, n_2, n_5 >$ and it goes through partial free node n_5. If (n_2, n_5) was zeroed at step

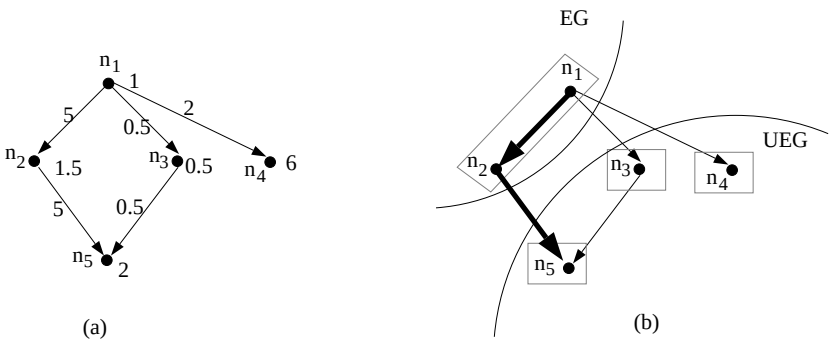


Figure 4.6: An example of a DS going through a partial free node n_5.

3, $tlevel(n_5)$ would have been decreased by $\delta = 5$. Then the length of the current DS $< n_1, n_2, n_5 >$ would also have been reduced by 5. But due to the topological traversal rule, a free task n_4 is selected at step 3 because $PRIO(n_4) = 9 \geq PRIO(n_3) = 4.5$. Then n_4 is merged to $CLUST(n_1)$ since $tlevel(n_4)$ can be reduced from 3 to $\tau_1 + \tau_2 = 2.5$. Such process affects the future compression of DS $< n_1, n_2, n_5 >$. When n_5 is free, $tlevel(n_5) = \tau_1 + \tau_2 + c_{2,5} = 7.5$ and it is impossible to reduce $tlevel(n_5)$ further by moving it to $CLUST(n_2)$. This is because n_5 will have to be linked after n_4, which makes $tlevel(n_5) = \tau_1 + \tau_2 + \tau_4 = 8.5$.

4.4 The Final Form of the DSC Algorithm

The main improvements to DSC-I are the minimization procedure, maintaining the partial free list and imposing the constraint DSRW.

4.4.1 Priority Lists

At each clustering step, we maintain two node priority lists, a partial free list PFL and a free list FL both sorted in a descending order of their task priorities. When two tasks have the same priority we break the tie by using the most immediate successors first (MISF) principle [57]. If the numbers of their successors are equal, we break the tie randomly. Function $head(L)$ returns the first node in the sorted list L, which is the task with the highest priority. If $L = \{\}$, $head(L) = NULL$ and the priority value is set to 0.

The *tlevel* value of a node is propagated to its successors only after this node has

been examined. Thus the priority value of a partial free node can be updated using only the *tlevel* from its examined predecessors. Because only part of predecessors are considered, we define the priority of a partial free task as

$$pPRIO(n_y) = ptlevel(n_y) + blevel(n_y)$$

where

$$ptlevel(n_y) = \max_{n_j \in PRED(n_y) \bigcap EG} \{tlevel(n_j) + \tau_j + c_{j,y}\}.$$

In general, $pPRIO(n_y) \leq PRIO(n_y)$ and if a DS goes through an edge (n_j, n_y) where n_j is one of the examined predecessors of n_y, then we have $pPRIO(n_y) = PRIO(n_j) = PRIO(n_y)$. By maintaining $pPRIO$ instead of $PRIO$ the complexity is reduced considerably. As we will prove later maintaining $pPRIO$ does not adversely affect the performance of DSC since it can still correctly identify the DS at each step. The DSC algorithm is described in Figure 4.7.

$EG = \emptyset$. $UEG = V$.
Compute *blevel* for each node and set $tlevel = 0$ for each free node.
WHILE $UEG \neq \emptyset$ **DO**
  $n_x = head(FL)$;/* the free task with the highest priority $PRIO$. */
  $n_y = head(PFL)$; /* the partial free task with the highest priority $pPRIO$.*/
  IF $(PRIO(n_x) \geq pPRIO(n_y))$ **THEN**
    Call the minimization procedure to reduce $tlevel(n_x)$.
    If no zeroing is accepted, n_x remains in a unit cluster .
  ELSE
    Call the minimization procedure to reduce $tlevel(n_x)$ under constraint DSRW.
    If no zeroing is accepted, n_x remains in a unit cluster .
  ENDIF
  Update the priority of n_x's successors and put n_x into EG.
ENDWHILE

Figure 4.7: The DSC Algorithm.

4.4.2 The minimization procedure for zeroing multiple incoming edges

To reduce $tlevel(n_x)$ in DSC a minimization procedure that zeros multiple incoming edges of n_x is needed. An optimal algorithm for a join DAG has been described in [42] and an optimal solution is shown in Figure 4.5(c). The basic procedure is to first sort

1. Sort the predecessors of n_x such that

$$tlevel(n_j) + \tau_j + c_{j,x} \geq tlevel(n_{j+1}) + \tau_{j+1} + c_{j+1,x}, \quad j = 1 : m-1.$$

2. Let h be the maximum integer from 2 to m such that for $2 \leq t \leq h$ node n_t satisfies the following constraint:

If n_t is not in $CLUST(n_1)$, then n_t does not have any children other than n_x.

3. Find the optimum point k between 1 and h using the binary search algorithm. Zero (n_1, n_x) up to (n_k, n_x) so that $tlevel(n_x)$ is minimum.

Figure 4.8: The minimization procedure.

the nodes such that $\tau_j + \beta_j \geq \tau_{j+1} + \beta_{j+1}, \quad j = 1 : m-1$. We then zero edges from left to right by using a *linear search* provided that the parallel time reduces after each zeroing. This is equivalent to satisfying the condition $(\sum_{j=1}^{k-1} \tau_j \leq \beta_k)$ for each accepted zeroing. Another optimum algorithm for join is to determine the optimum point k first by using a *binary search* between 1 and m such that $(\sum_{j=1}^{k-1} \tau_j \leq \beta_k)$ and then zero all edges to the left of k.

We will adapt this optimum join algorithm to minimize $tlevel(n_x)$ for DSC. Assume that $PRED(n_x) = \{n_1, n_2, \cdots, n_m\}$. Notice that all predecessors are in EG but now $CLUST(n_1), \cdots, CLUST(n_m)$ might not be unit clusters. We sort the predecessors of n_x such that $tlevel(n_j) + \tau_j + c_{j,x} \geq tlevel(n_{j+1}) + \tau_{j+1} + c_{j+1,x}$. An optimum algorithm will zero the edge (n_1, n_x) first. A problem arises when the algorithm zeroes an edge of a predecessor n_p which has other children than n_x. The task n_p will be *extracted* from $CLUST(n_p)$ and be moved to $CLUST(n_1)$. As a result the *tlevel* of the children of n_p will be affected and the length of paths going through those children will most likely be increased. The constraint $PT_{i-1} \geq PT_i$ may no longer hold and maintaining task priorities becomes complicated. Therefore we exclude from the minimization procedure predecessors which have children other than n_x with few exceptions. Figure 4.8 is the algorithm of this minimization procedure.

Figure 4.8 uses the binary search algorithm to find the best stopping point k and those predecessors of n_x that are not in $CLUST(n_1)$ must be extracted from their corresponding clusters and attached to $CLUST(n_1)$. Those attached predecessors are

ordered for execution in an increasing order of their *tlevel* values.

We determine the complexity of this procedure. The *tlevel* ordering of all predecessors is done once at a cost of $O(m \log m)$. The binary search computes the effect of the ordered predecessors to $tlevel(n_x)$ at a cost of $O(m)$ for $O(logm)$ steps. The total cost of the above algorithm is $O(m \log m)$. If linear search was used instead the total cost will increase to $O(m^2)$.

4.4.3 Imposing constraint DSRW

As we saw previously when there is no DS going through any free task and there is one DS passing through a partial free node n_y, then zeroing non-DS incoming edges of free nodes could affect the reduction of $tlevel(n_y)$ in the future steps. We impose the following constraint on DSC to avoid such side-effects:

- DSRW: Zeroing incoming edges of a free node should not affect the reduction of $ptlevel(n_y)$ if it is reducible when zeroing an incoming edge of n_y.

There are two problems that we must address in the implementation of DSRW. First we must detect that $ptlevel(n_y)$ is reducible and second we must make sure that DSRW is satisfied.

1. To detect the reducibility of $ptlevel(n_y)$ we must examine the result of the zeroing of an incoming DS edge to n_y. To find such an incoming DS edge we only examine the result of the zeroing each incoming edge (n_j, n_y) where n_j is a predecessor of n_y and $n_j \in EG$. As we will prove in section 4.4.5 $ptlevel(n_y) = tlevel(n_y)$. This implies that such partial reducibility suffices to guarantee that if the parallel time was reducible by zeroing the DS incoming edges of a partial free DS node n_y, then $tlevel(n_y)$ is reducible when n_y becomes free. Hence the DS can be compressed at that time.

2. After detecting the partial reducibility at step i for node n_y, we implement the constraint DSRW as follows: Assume that n_p is one examined predecessor of n_y and zeroing (n_p, n_y) would reduce $ptlevel(n_y)$, then no other nodes are allowed to move to $CLUST(n_p)$ until n_y becomes free.

For the example in Figure 4.6(b), $n_y = n_5$, $ptlevel(n_5) = \tau_1 + \tau_2 + c_{2,5} = 7.5$, $pPRIO(n_5) = 9.5$, $PRIO(n_3) = 4.5$ and $PRIO(n_4) = 9$. We have that $pPRIO(n_5) > PRIO(n_4)$, which implies DS goes through partial free node n_5 by Theorem 4.1 in section 4.4.5. And $ptlevel(n_5)$ could be reduced if (n_2, n_5) was zeroed. Then $CLUST(n_2)$ cannot be touched before n_5 becomes free. Thus n_3 and n_4 cannot be moved to $CLUST(n_2)$ and they remain in the unit clusters in EG. When finally n_5 becomes free, (n_2, n_5) is zeroed and PT is reduced to 9.

4.4.4 A running trace of DSC

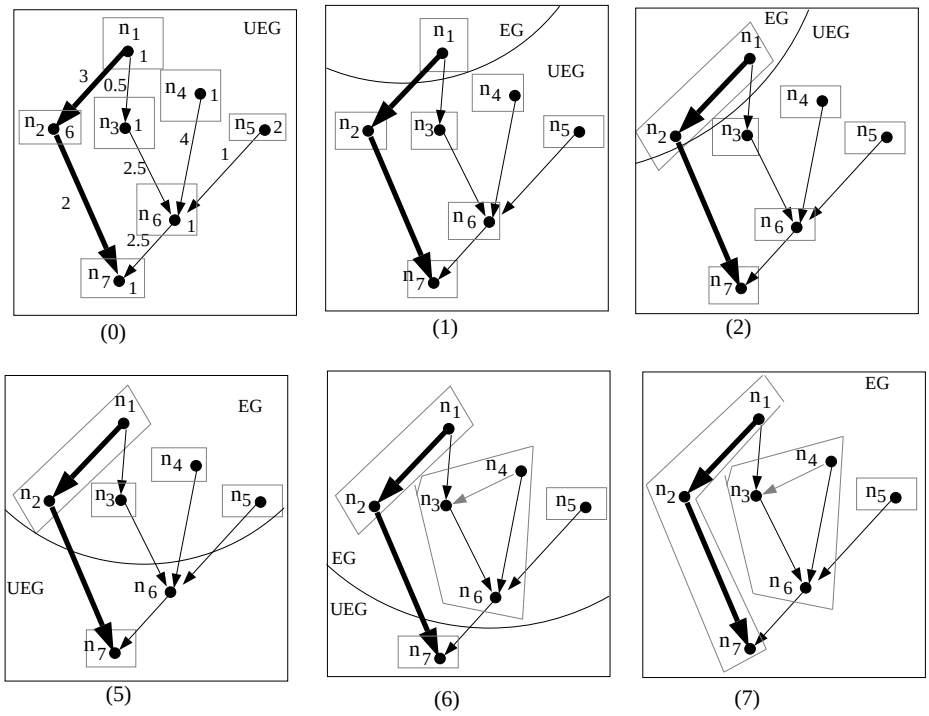


Figure 4.9: DSC clustering steps for the DAG shown in Figure 1(a).

We demonstrate the DSC steps by using a DAG example shown in Figure 4.1(a). The result of each DSC clustering step (initial, step 1, 2, 5, 6, final) is shown in Figure 4.9. The thick paths of each graph represent DSs and dashed edges pseudo execution edges. We explain each step below. The superscript of a task node in FL or PFL indicates its priority value.

- Initially, $USG = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$, $PT_0 = 13$,

$FL = \{n_1^{0+13}, n_4^{0+9.5}, n_5^{0+7.5}\}$, $PFL = \{\}$.

- Step 1, n_1 is selected, $tlevel(n_1) = 0$, it cannot be reduced so $CLUST(n_1)$ remains a unit cluster. Then $USG = \{n_2, n_3, n_4, n_5, n_6, n_7\}$, $PT_1 = 13$, $FL = \{n_2^{4+9}, n_3^{1.5+8}, n_4^{0+9.5}, n_5^{0+7.5}\}$, $PFL = \{\}$.

- Step 2, n_2 is selected, $n_x = n_2$, $n_y = NULL$, and $tlevel(n_2) = 4$. By zeroing the incoming edge (n_1, n_2) of n_2, $tlevel(n_2)$ reduces to 1. Thus this zeroing is accepted and after that step, $USG = \{n_3, n_4, n_5, n_6, n_7\}$, $PT_2 = 10$, $FL = \{n_3^{1.5+8}, n_4^{0+9.5}, n_5^{0+7.5}\}$, $PFL = \{n_7^{9+1}\}$.

- Step 3, n_3 is selected, $n_x = n_3$ with $PRIO(n_3) = 1.5 + 8 = 9.5$ and $n_y = n_7$ with $pPRIO(n_7) = 10$. Zeroing (n_2, n_7) would reduce $pPRIO(n_7)$ and DSRW is imposed. For n_3, zeroing the incoming edge (n_1, n_3) is excluded for the consideration because of DSRW. The $tlevel(n_3)$ remains the same and $CLUST(n_3)$ remains a unit cluster in EG. At step 4 and 5, n_4 and n_5 are selected respectively and their clusters remains unit clusters also. After that, $USG = \{n_6, n_7\}$, $PT_5 = 10$, $FL = \{n_6^{5+4.5}\}$, $PFL = \{n_7^{9+1}\}$.

- Step 6, n_6 is selected and its incoming edges (n_3, n_6) and (n_4, n_6) are zeroed by the minimization procedure. Node n_4 is ordered for execution first because $tlevel(n_4) = 0$, which is smaller than $tlevel(n_3) = 1.5$. Then $tlevel(n_6)$ is reduced to 2.5 and $USG = \{n_7\}$, $PT_6 = 10$, $FL = \{n_7^{9+1}\}$, $PFL = \{\}$.

- Step 7, n_7 is selected and its incoming edge (n_2, n_7) is zeroed so that $tlevel(n_7)$ is reduced from 9 to 7. $USG = \{\}$, $PT_7 = 8$, $FL = \{\}$, $PFL = \{\}$.

Finally three clusters are generated with PT=8.

4.4.5 DSC Properties

In this section, we study several properties of DSC. Theorem 4.1 indicates that the DSC algorithm correctly locates DS nodes at each step even after we use the partial priority $pPRIO$ for partial free nodes. Theorems 4.2 and 4.3 examine the effort made

by DSC in reducing the parallel time. Theorem 4.4 shows that the complexity of DSC is $O((v+e)\log v)$. Notice that for most of program dependence graphs, e is in the same order of v.

The correctness in locating DS nodes

Lemma 4.1 *Assume that $n_x = head(FL)$ after step i. If there are DSs going through free nodes in UEG, then one DS must go through n_x.*

Proof: Let the current parallel time be PT_i. Each free node n_s in DSs satisfies $PT_i = PRIO(n_s)$. If no DS goes through n_x but one DS goes through another non-head free node n_f, then $PRIO(n_f) = PT_i > PRIO(n_x)$. This indicates that n_x is not the head of FL, which is a contradiction. □

Lemma 4.2 *Assuming $n_y = head(PFL)$ after step i. If DSs only go through partial free nodes in UEG, then one DS must go through n_y. Moreover, $pPRIO(n_y) = PRIO(n_y)$.*

Proof: The starting node of a DS must be an entry node of this DAG. If it is in UEG, this node must be free, which is impossible. Thus a DS must start from a node in EG and go through an examined node n_j to its unexamined partial free successor n_p. The $pPRIO(n_p)$ value is propagated from n_j and $PRIO(n_p) = PRIO(n_j) = pPRIO(n_p)$. Suppose no DS goes through n_y, we have

$$PRIO(n_y) < PT_i = PRIO(n_p) = pPRIO(n_p).$$

Since $pPRIO(n_y) \leq PRIO(n_y)$, $pPRIO(n_y) < pPRIO(n_p)$, n_y is not the head of PFL, which is a contradiction.

Next we show $pPRIO(n_y) = PRIO(n_y)$. Suppose $pPRIO(n_y) < PRIO(n_y)$, which indicates that the DS where n_y resides does not pass through any examined immediate predecessor of n_y. This DS must go through some other nodes in UEG. Then we should be able to find an ancestor node n_a of n_y, satisfying the following condition: the DS passes an edge (n_q, n_a) where n_q is the examined predecessor of n_a and n_a is partial

free since it is impossible to be free. Thus

$$pPRIO(n_a) = PRIO(n_a) = PRIO(n_y) > pPRIO(n_y)$$

which shows n_y is not the head of PFL. This a contradiction. □

Theorem 4.1 *Assume that $n_x = head(FL)$ and $n_y = head(PFL)$ after step i and there is a DS going through UEG. If $PRIO(n_x) \geq pPRIO(n_y)$, then a DS goes through n_x. If $PRIO(n_x) < pPRIO(n_y)$, then DS does not go through any free node and it passes through n_y.*

Proof: When there is a DS going through UEG, if $PRIO(n_x) \geq pPRIO(n_y)$, a DS must pass through a free node otherwise according to Lemma 4.2, a DS is supposed to go through n_y which is a contradiction since then

$$pPRIO(n_y) = PRIO(n_y) = PT_i > PRIO(n_x).$$

If $PRIO(n_x) < pPRIO(n_y)$, suppose that a DS passes a free node, then according to Lemma 4.1,

$$PRIO(n_x) = PT_i \geq PRIO(n_y) \geq pPRIO(n_y)$$

which is a contradiction again. Thus the DSs must go through partial free nodes and one of them must go through n_y by Lemma 4.2. □

The warranty in reducing parallel time

Lemma 4.3 *Assume that $n_x = head(FL)$ and $n_y = head(PFL)$ after step i. The parallel time for executing the clustered graph after step i of DSC is:*

$$PT_i = \max\{PRIO(n_x),\ pPRIO(n_y),\ \max_{n_e \in EG}\{PRIO(n_e)\}\}.$$

Proof: There are three cases in the proof:

1. If DS nodes are only within EG, according to Equation (4.1), $PT_i = \max_{n_e \in EG}\{PRIO(n_e)\}$.

2. If a DS goes through a free node, then $PT_i = PRIO(n_x)$ by Lemma 4.1.

3. If there is a DS passing through UEG but this DS only passes through partial free nodes, then $PT_i = PRIO(n_y) = pPRIO(n_y)$ by Lemma 4.2 . □

When all DSs are in EG, those DS edges are not considered for re-examination because of the non-backtracking assumption. DSC continues to compress a task sequence whose length is the second highest, third, and so forth, to eliminate unnecessary processors.

When one DS goes through UEG, by Lemma 4.3 the overall parallel time could be reduced by decreasing $PRIO(n_x)$ if a DS is passing n_x.

Theorem 4.2 *For each step i of DSC, $PT_{i-1} \geq PT_i$.*

Proof: When DSC zeroes incoming edges of a free node n_x, the priority values of its immediate predecessors and all descendant may be changed. The minimization procedure may increase the *tlevel* values of some predecessors of n_x, but those predecessors have no children other than n_x and $tlevel(n_x)$ does not increase. The length of paths going through n_x decreases or remains unchanged. Thus the priority values of n_x's predecessors may increase, but those new values still are less than the new value of $PRIO(n_x)$. Also the priority values of descendants of n_x could decrease but not increase.

When edge zeroing adds n_x to a cluster in EG, a pseudo edge is added from the last node of that cluster, say n_e, to n_x if n_e and n_x are independent. Then $PRIO(n_e)$ may increase since $blevel(n_e)$ may increase. If this is the case, then we can prove that after such an increase, $PRIO(n_e) \leq PRIO(n_x)$.

Thus by Lemma 4.3, DSC tries to reduce the priority of n_x and not increase it and as a result PT either decreases or remains unchanged. □

Theorem 4.3 *After step i of DSC, if the current parallel time is reducible by zeroing one incoming edge of a node in UEG, then DSC guarantees that either the parallel time is reduced at step $i+1$ or it will be reduced at some step later.*

Proof: The assumption of this theorem implies that a DS must go through UEG. Suppose that zeroing an edge (n_r, n_s), where $n_s \in UEG$, will reduce $tlevel(n_s)$ and also

the current parallel time. Then all DSs go through that edge. The DS nodes will only be n_r, n_s, some ancestors of n_r and some descendants of n_s. There are three cases:

1. $n_r \in EG$ and n_s is free.

   n_s must be the head of FL since other DS nodes in UEG must be descendants of n_s and cannot be free. Then n_s is picked up at step $i+1$. Since $tlevel(n_s)$ is reducible by zeroing (n_r, n_s), the minimization procedure will zero this edge and PT decreases.

2. $n_r \in EG$ and n_s is partial free.

   No free nodes are in DSs and n_s must be the head of PFL. Other partial free nodes, say n_f, could be in DSs but $pPRIO(n_f) < pPRIO(n_s)$.

   At step $i+1$, the reducibility of $ptlevel(n_s)$ is detected. Before n_s becomes free, PT remains unchanged, a non-DS node remains to be a non-DS node and no other nodes are moved to $CLUST(n_r)$. When n_s becomes free, (n_r, n_s) is zeroed to reduce $tlevel(n_s)$ at that step and PT should be reduced if it was reducible after step i.

3. $n_r \in UEG$.

   $tlevel(n_s)$ remains reducible until some step j when n_r becomes examined. If the PT has not been reduced from step $i+1$ to j, one DS that (n_s, n_s) lies must remain unchanged and PT is still reducible by zeroing (n_r, n_s) at step j. Then the scenario becomes exactly the same as in Case 1 or 2. □

Corollary 4.3 *Assume that n_y is a partial free node at step i and it is in DS and that the zeroing of an incoming edge of n_y from a scheduled predecessor would have reduced PT by δ. Then DSC guarantees that when n_y becomes free at step j ($j > i$), PT can be reduced by at least δ.*

Proof: At step i, $tlevel(n_y)$ is reducible. By Theorem 4.3, DSC detects the reducibility and imposes constraint DSRW. At step j, n_y is free and incoming edges of n_y can be zeroed. From step i to step j, edge zeroing compresses a SubDS. If such zeroing adds a free node to a cluster and this cluster becomes nonlinear, a pseudo edge is added

and the length of all paths going though n_x will not increase. At step i, zeroing some incoming edge of n_y from a scheduled predecessor could reduce PT by δ. Then after step j, SubDSs have been compressed further and PT will be reduced by at least δ. □

The complexity

We now perform the complexity analysis.

Lemma 4.4 *For any node n_s in FL or PFL, $tlevel(n_j)$ for $n_j \in PRED(n_s) \bigcap EG$ remains constant after n_j is examined, and $blevel(n_s)$ remains constant until it is examined.*

Proof: Referring to Property 4.2 of DSC-I, DSC is the same as DSC-I except that the minimization procedure changes *tlevel* values of some examined predecessors, say n_h, of the currently-selected free task, say n_x. But such change does not affect any node in FL or PFL since n_h does not have children other than n_x. □

Theorem 4.4 *The time complexity of DSC is $O((v+e)\log v)$ and the space complexity is $O(v+e)$.*

Proof: The major cost of DSC is in maintaining two priority lists and in executing the minimization procedure. When a selected free node n_x is moved from UEG to EG, the *tlevel* value of each of its successors is updated. The cost of updating the FL and PFL will be $O(|SUCC(n_x)\log v|)$ where $O(\log v)$ is the cost for adjusting the order of a priority list. According to lemma 4.4, after n_x has moved to EG, its *tlevel* will not affect the priority of nodes in PFL and FL. Adding the cost to delete the head of FL at each step and to move nodes from PFL to FL, the total cost of maintaining FL and PFL is $((e+v)\log v)$.

The minimization procedure costs $O(|PRED(n_x)|\log|PRED(n_x)|)$ at each step and for v nodes. The overall cost is $O(e\log v)$.

Thus the overall time complexity is $O((e+v)\log v)$. The space needed for DSC is to store the DAG and FL/PFL. The space complexity is $O(v+e)$. □

4.5 Performance Bounds and Optimality of DSC

In this section we study the performance bound of DSC for a general DAG and the optimality for special classes of DAGs.

4.5.1 Performance bounds for general DAGs

In Chapter 3 we have discussed a definition of granularity for a DAG. Based on this definition we now give a performance bound of DSC for a general DAG.

Theorem 4.5 *Let PT_{dsc} be the parallel time by DSC for a DAG G, then $PT_{dsc} \leq (1 + \frac{1}{g(G)})PT_{opt}$. For a coarse grain DAG, $PT_{dsc} \leq 2 \times PT_{opt}$.*

Proof: In the initial step of DSC, all nodes are in the separate clusters, which is a linear clustering. By Theorem 3.2 and Theorem 4.2 we have that

$$PT_{dsc} \leq \ldots \leq PT_k \leq \ldots \leq PT_1 \leq PT_0 \leq (1 + \frac{1}{g(G)})PT_{opt}.$$

For coarse grain DAGs the statement is obvious. □

The above theorem provides an upper bound for DSC. For example, if the partitioning is such that $g(G) > 10$ then the DSC will be within 90% of the optimum. This theorem also implies that low complexity algorithms such as DSC are sufficient for clustering coarse grain task graphs. On the other hand the bound is not sharp for fine grain task graphs and an open question remains if a sharper upper bound can be found for DSC.

4.5.2 Optimality for join and fork

Theorem 4.6 *DSC derives optimal solutions for fork and join DAGs.*

Proof: For a fork, DSC performs the exact same zeroing sequence as DSC-I. The clustering steps are shown in Figure 4.4. After n_x is examined, DSC will examine free nodes $n_1, n_2, \cdots, n_m$ in a decreasing order of their priorities. Note that the priority value for each free node is the length of each path $< n_x, n_1 >$, $\cdots$, $< n_x, n_m >$. If we

assume that $\beta_k + \tau_k \geq \beta_{k+1} + \tau_{k+1}$ for $1 \leq k \leq m-1$, then the nodes are sorted as n_1, $n_2, \cdots, n_m$ in the free list FL.

We now determine the optimal time for the fork and then show that DSC achieves the optimum. Assume the optimal parallel time to be PT_{opt}. If $PRIO(n_h) = \tau_x + \tau_h + \beta_h > PT_{opt}$ for some h, then β_i must be zeroed for $i = 1 : h$, otherwise we have a contradiction. Also $\beta_i (i > h)$ need not be zeroed because zeroing such edge does not decrease PT but could increase PT.

Let the optimal zeroing stopping point be h and assume $\beta_{m+1} = \tau_{m+1} = 0$. Then the optimal PT is:

$$PT_{opt} = \tau_x + max(\sum_{j=1}^{h} \tau_j, \beta_{h+1} + \tau_{h+1}). \tag{4.2}$$

DSC zeroes edges from left to right as many as possible up to the point k as shown in Figure 4.4(d) such that:

$$\sum_{j=1}^{k-1} \tau_j \leq \beta_k \qquad \text{and} \qquad \sum_{j=1}^{k} \tau_j > \beta_{k+1}.$$

We will show $PT_{opt} = PT_{dsc}$ by contradiction. Suppose that $k \neq h$ and $PT_{opt} < PT_{dsc}$. There are two cases:

If $h < k$, then

$$\sum_{j=1}^{h} \tau_j < \sum_{j=1}^{k} \tau_j \leq \beta_k + \tau_k \leq \beta_{h+1} + \tau_{h+1}.$$

Thus Equation (4.2) can be simplified further as:

$$PT_{opt} = \tau_x + \beta_{h+1} + \tau_{h+1} \geq \tau_x + \beta_k + \tau_k \geq \tau_x + max(\sum_{j=1}^{k} \tau_j, \beta_{k+1} + \tau_{k+1}) = PT_{dsc}.$$

If $h > k$, then since $\sum_{j=1}^{h} \tau_j \geq \sum_{j=1}^{k+1} \tau_j > \beta_{k+1} + \tau_{k+1} \geq \beta_{h+1} + \tau_{h+1}$,

$$PT_{opt} = \tau_x + \sum_{j=1}^{h} \tau_j \geq \tau_x + max(\sum_{j=1}^{k} \tau_j, \beta_{k+1} + \tau_{k+1}) = PT_{dsc}.$$

There is a contradiction in both cases.

For a join, the DSC uses the minimization procedure to minimize the *tlevel* value of the root and the solution is symmetrical to the optimal result for a fork. □

Chretienne [20] describes a special algorithm for scheduling a fork with a complexity of $O(m \log B)$ where and $B = min\{\sum_{i=1}^{m} \tau_i, \beta_1 + \tau_1\} + \tau_x$. For a comparison the DSC costs $O(m \log m)$ for the fork.

4.5.3 Optimality for in/out trees

An *in-tree* is a directed tree in which the root has outgoing degree zero and other nodes have the outgoing degree one. An *out-tree* is a directed tree in which the root has incoming degree zero and other nodes have the incoming degree one.

Scheduling in/out trees is still NP-hard in general as shown by Chretienne [22] and DSC will not give the optimal solution. However, DSC will yield optimal solutions for coarse grain trees and a class of fine grain trees.

Coarse grain trees

Theorem 4.7 *DSC gives an optimal solution for a coarse grain in-tree.*

Proof: Since all paths in an in-tree go through the tree root, say n_x, $PT = tlevel(n_x) + blevel(n_x) = tlevel(n_x) + \tau_x$. We claim $tlevel(n_x)$ has the minimum value. We prove it by induction on the depth of the in-tree (d).

When $d = 0$, it is trivial. When $d = 1$, it is a join DAG and $tlevel(n_x)$ is minimized.

Assume it is true for $d < k$.

When $d = k$, let the predecessors of root n_x be $n_1, \cdots, n_m$. Since each sub-tree rooted with n_i has depth $< k$ and the disjoint subgraphs cannot be clustered together by DSC, DSC will obtain the minimum $tlevel$ time for each n_j where $1 \leq j \leq m$ according to the induction hypothesis.

When n_x becomes free, its $tlevel$ is

$$tlevel(n_x) = \max_{1 \leq j \leq m} \{tlevel(n_j) + \tau_j + c_{j,x}\}.$$

When n_x is selected, $tlevel(n_x)$ is minimized by DSC. Without loss of generality, assume that (n_1, n_x) is in a DS and $tlevel(n_1) + \tau_1 + c_{1,x}$ has the highest value and $tlevel(n_2) + \tau_2 + c_{2,x}$ has the second highest value. To make an expression short, we define $CT(n_j) = tlevel(n_j) + \tau_j$.

DSC will zero (n_1, n_x), which will decrease $tlevel(n_x)$ as follows

$$tlevel(n_x) = \max(CT(n_1), \max_{2 \leq j \leq m} \{CT(n_j) + c_{j,x}\}). \tag{4.3}$$

DSC will not zero any more edge because of the coarse grain condition ($g(G) > 1$). DSC might zero (n_2, n_x) when $g(G) = 1$, but $tlevel(n_x)$ does not decrease any further. Thus we can use Equation (4.3) as the *tlevel* value of root n_x when the tree depth is $h = k$.

To prove $tlevel(n_x)$ is the smallest, we need to compare with an optimal schedule. Since the tree is coarse-grain, by Theorem 3.3, linear clustering can be used for deriving the optimal solution. Thus we can assume S^* is an optimal schedule that uses linear clustering. Let $tlevel^*(n_j)$ be the *tlevel* value of n_j in schedule S^*, and $CT^*(n_j) = tlevel^*(n_j) + \tau_j$, $CT^*(n_j) \geq CT(n_j)$ for $1 \leq j \leq m$ according the above result. Now we will show that $tlevel^*(n_x) \geq tlevel(n_x)$. There are two cases:

Case 1) If in S^* the zeroed incoming edge of n_x is (n_1, n_x), then

$$tlevel^*(n_x) = \max(CT^*(n_1), \max_{2 \leq j \leq m} \{CT^*(n_j) + c_{j,x}\}) \geq \max(CT(n_1), \max_{2 \leq j \leq m} \{CT(n_j) + c_{j,x}\}).$$

Thus $tlevel^*(n_x) \geq tlevel(n_x)$.

Case 2) If in S^* the zeroed incoming edge of n_x is not (n_1, n_x), say it is (n_m, n_x). Thus

$$\begin{aligned} tlevel^*(n_x) &= \max(CT^*(n_m), \max_{1 \leq j \leq m-1} \{CT^*(n_j) + c_{j,x}\}) \\ &\geq \max(CT(n_m), \max_{1 \leq j \leq m-1} \{CT(n_j) + c_{j,x}\}). \end{aligned}$$

Because of $g(G) \geq 1$,

$$CT(n_1) + c_{1,x} \geq \max_{2 \leq j \leq m} \{CT(n_j) + c_{j,x}\},$$

then

$$tlevel^*(n_x) \geq CT(n_1) + c_{1,x} \geq tlevel(n_x).$$

That shows $tlevel(n_x)$ is optimal for a coarse grain in-tree with $d = k$. □

DSC solves an in-tree in time $O(v \log v)$ where v is the number of nodes in this tree. Chretienne [21] developed an $O(v)$ optimal algorithm for this tree. Anger, Hwang and Chow [6] proposed an $O(v)$ optimal algorithm for tree DAGs with the condition that all communication edge weights are smaller than the task node execution weights, which

are special cases of coarse grain tree DAGs. These two algorithms are only specific to this kind of trees.

For an out-tree, we can always inverse this DAG and use DSC to schedule the inverted graph. Then the optimal solution can be obtained. We call this approach the backward dominant sequence clustering [5].

Fine grain trees

Finding optimal solutions for general fine grain trees is NP-complete. However, DSC is able to obtain optimum for a class of fine grain trees. A *single-spawn* out-tree is an out tree such that at most one successor of a non-leaf tree node, say n_x, can spawn successors. Other successors of n_x are leaf nodes. A *single-merge* in-tree is an inverse of a single-spawn out tree. Examples of such trees are shown in Figure 4.10.

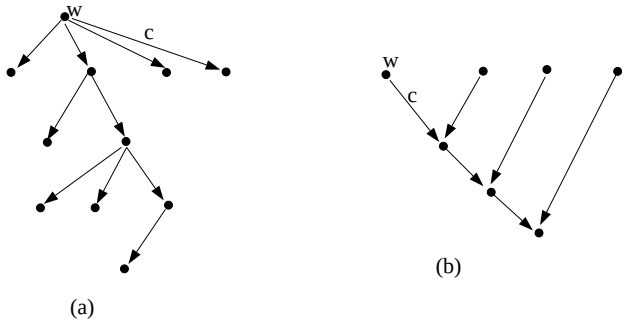


Figure 4.10: (a) A single-spawn out-tree. (b) A single-merge in-tree.

Theorem 4.8 *Given a single-spawn out-tree or single-merge in-tree with an equal computation weight w for each task and an equal communication weight c for each edge, DSC is optimal for this tree.*

Proof: We will present a proof for an out-tree by induction on the height (h) of this tree. The proof for an in-tree is similar.

When $h = 2$, it is a fork and DSC is optimal. Assume DSC obtains the optimum when $h = k$.

[5] In the actual DSC program, both backward and forward clusterings are performed and the one with the better solution is chosen.

When $h = k + 1$, we assume without loss of generality that the successors of root n_0 are $n_1, n_2, \cdots, n_j$ and that n_1 spawns successors. First we assume n_0 has more than 1 successors, i.e. $j > 1$. Figure 4.11 depicts this tree. We call the entire tree T^{k+1} and the subtree rooted by n_1 T^k. The height of T^k is k and it has q tasks where $q > 1$.

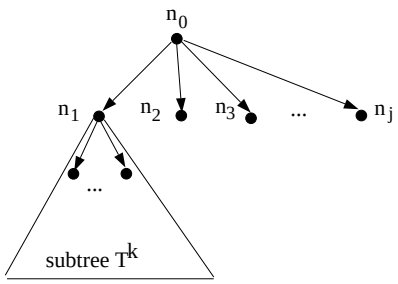


Figure 4.11: A single-spawn out-tree named T^{k+1} with height $h = k + 1$.

We claim that DSC will examine all nodes in subtree T^k first before examining other successors of n_0. At step 1, n_0 is examined and all successors of n_0 become free. Node n_1 has priority $PRIO(n_1) \geq 3w + 2c$ and other successors of n_0 have priority $2w + c$. Then n_1 is examined at step 2, (n_0, n_1) is zeroed and all successors of n_1 are added to the free list. The priority of n_1's successors $\geq 3w+c$. Thus they will be examined before $n_2, \cdots, n_j$. Recursively, all n_1's descendants will be freed and have priority $\geq 3w + c$. Thus from step 2 to step $q + 1$, all q nodes in T^k are examined one by one. After step $q + 1$, DSC looks at $n_2, n_3, \cdots, n_j$.

Since from step 2 to $q+1$, DSC clusters T^k only, DSC obtains an optimal clustering solution for this subtree by the induction hypothesis. We call the parallel time for this subtree $PT_{opt}(T^k)$. Then the parallel time after step $q + 1$ is:

$$PT_{dsc}^{q+1} = \max(w + PT_{opt}(T^k), 2w + c).$$

Let PT_{dsc} be the time after the final step of DSC for T^{k+1}. We study the following two cases in clustering T^k by DSC.

1. One edge in T^k is not zeroed by DSC. Then $PT_{opt}(T^k) \geq 2w + c$ implying $PT_{dsc}^{q+1} = w + PT_{opt}(T^k)$. Let PT_{dsc} be the time of the final step, since the step-wise parallel time of DSC monotonically decreases, $PT_{dsc}^{q+1} \geq PT_{dsc}$. Also since the optimal parallel time for a graph should be no less than that for its subgraph,

$PT_{opt}(T^{k+1}) \geq w + PT_{opt}(T^k)$. Thus $PT_{dsc} \leq PT_{opt}(T^{k+1})$ and DSC is optimal for this case.

2. All edges in T^k are zeroed by DSC. Then $PT_{opt}(T^k) = qw$ and $PT_{dsc}^{q+1} = \max(w + qw, 2w + c)$. If $w + qw \geq 2w + c$, i.e., $c \leq (q-1)w$ then

$$PT_{dsc} \leq PT_{dsc}^{q+1} = w + PT_{opt}(T^k) \leq PT_{opt}(T^{k+1}).$$

Since otherwise $c > (q-1)w$ and $PT_{dsc}^{q+1} = 2w + c$.

We claim that all edges in T^k and edge (n_0, n_1) should be zeroed by any optimal clustering for T^{k+1}. If they are not, then $PT_{opt}(T^{k+1}) \geq 3w + c > PT_{dsc}^{q+1} \geq PT_{dsc}$, which is impossible.

Since all nodes in T^k and n_0 are in the same cluster, the optimal clustering for the entire out-tree T^{k+1} can be considered as clustering a fork with "leaf-node" n_1 having a weight qw. Because DSC is optimal for a fork, DSC will get the optimum for T^{k+1}.

Finally we examine the case that n_0 only has one successor, i.e. $j = 1$. DSC first zeros edge (n_0, n_1) then gets the optimum for T^k. Thus $PT_{dsc} = w + PT_{opt}(T^k) = PT_{opt}(T^{k+1})$. □

We do not know of another proof of polynomiality of the above class of fine grain DAGs in the literature. An open question remains if there exist a larger class of fine grain trees which are tractable in polynomial time, say for example the weights are not uniform in the above trees.

4.6 A Comparison with Other Algorithms and Experimental Results

There are other algorithms for general DAGs developed by Sarkar [90], Kim and Browne [60], and Wu and Gajski [98]. Kim and Browne's produces linear clusterings. Wu and Gajski's MCP algorithm is similar to DSC-I except that the priority of a free node is chosen as *blevel* instead of *tlevel* + *blevel*. A comparison of these algorithms is given in Gerasoulis and Yang [44]. Table 4.2 summaries a comparison of these algorithms with DSC on complexity and optimality results.

Sarkar	MCP	Kim &Browne	DSC
$O(e(v+e))$	$O(v^2 \log v)$	$O(v(v+e))$	$O((v+e)\log v)$

	Sarkar	MCP	Kim &Browne	DSC
Join/Fork	no	no	no	optimal
Coarse grain tree	no	optimal	no	optimal
Fine grain tree single spawn/merge	no	no	no	optimal

Table 4.2: A comparison of clustering algorithms.

Since this scheduling problem is NP-complete for a general fine grain tree and for a DAG which is a concatenation of a fork and a join(series parallel DAG) [20, 22], the analysis result shows that DSC not only has a low complexity but also attains an optimality degree as much as a general polynomial algorithm could achieve.

In [105], we have also compared DSC with the MD algorithm [98] and the ETF algorithm [54]. The idea of identifying the important tasks (DS nodes) in DSC is the same as MD. However, the way to find DS nodes is different. DSC uses a priority function and embodies a $O(\log v)$ computing scheme while MD uses the relative mobility function with a computing cost of $O(v+e)$. The overall complexity of MD is $O(v^3)$. Another difference is that when DSC picks a DS task n_p to schedule, it uses the minimization procedure to reduce the *tlevel* of this task and thus decrease the length of DS going through this task, while MD scans the processors from the left to right to find the first processor as long as it satisfies a condition called Fact 1[6].

ETF [54] is a scheduling algorithm for a bounded number (p) of processors with arbitary network topologies with a complexity of (pv^2). ETF performs v scheduling step and selects a task with the eariest starting time at each step for scheduling. The task prioritization in ETF differs from DSC since a task with the eariest starting time may not be in a DS. Another difference is that ETF does not use the minimization procedure. A detailed comparison and experiment results will be in [105].

Next we describe our experiments and compare with other algorithms in scheduling

[6]Fact 1 in MD tries to guarantee the non-increasing of the length of the current critical path but it does not necessarily make the length shorter. In a recent personal communication [99], the authors of MD made some corrections for the Fact 1 description in [98], pp. 336. See [105] for the detail.

DAGs other than the primitive structures.

4.6.1 Random DAGs

	#cases	#tasks Min-Max	#tasks Ave	#edges Ave	C/R Min-Max	PT_Impro Min-Max	PT_Impro Ave	Time Faster
G1	22	44-98	70.1	311.3	0.83-5.6	4.6%-33.5%	17.8%	92.5
G2	47	103-198	148.7	502.2	0.27-7.6	3.3%-31%	21.6%	229.6
G3	31	250-540	329.0	3430	1.68-8.7	21%-37.8%	25.8%	1854.5

Table 4.3: The improvement of DSC over Sarkar's on random graphs.

We have run Sarkar's algorithm and the DSC algorithm using 100 randomly generated DAGs. A random DAG is generated by first randomly generating the number of nodes, then randomly linking the edges between them and finally assigning random values to node and edge weights. The results are summarized in Table 4.3. The explanation for each column is described as follows: In column 1, we have categorized task graphs into three groups in terms of their node sizes: G1: $v < 100$, G2: $100 \leq v < 200$ and G3: $v \geq 200$. Column 2 is the number of cases for each group. Column 3 is the minimum and maximum of the value v for each randomly generated case group. Column 4 is the average value of v. Column 5 is the average value of edge number e. Column 6 is the minimum and maximum values of C/R where C/R is the ratio of communication over computation, which we consider as an estimation of the inverse of the granularity $1/g$ of the graph. It is measured by dividing the total communication delay in the critical path by the total computation in the critical path. Column 7 is the minimum and the maximum of the parallel time improvement ratio. Column 8 is the average value. The improvement ratio of DSC over algorithm A is defined as

$$PT\_Improv = 1 - \frac{PT_{dsc}}{PT_A}.$$

Column 9 shows how much faster DSC is over Sarkar's for tested cases, which is the ratio between the computing time spent by Sarkar's algorithm and the time spent by DSC.

This experiment shows that the parallel time improvement ratio of DSC over Sarkar's

algorithm is about 20%. However, in terms of computing cost, DSC is superior particularly for large graphs. For a DAG with hundreds of nodes, it is shown that DSC is 1854 faster than Sarkar's on average. For such kind of DAGs, DSC takes few seconds to schedule while Sarkar's algorithm takes hours. This is expected because of the complexities of the algorithms, $O(e(v+e))$ for Sarkar's vs. $O((v+e)\log v)$ for DSC.

4.6.2 Choleski Decomposition DAG

In the second example we use a well known numerical computing DAG, the Choleski decomposition (CD) DAG given in Cosnard et al. [26]. In Figure 4.12 we show the PT improvement ratio (PT_Improv) of Kim and Browne's and MCP over DSC. The matrix is of dimension 250×250 and $v = 31000$ nodes and $e = 62000$ edges. The x-axis is the inverse granularity ratio $1/g$ of the graph. The larger the $1/g$ the finer the granularity of the DAG, and the finest granularity is when $1/g = 50$.

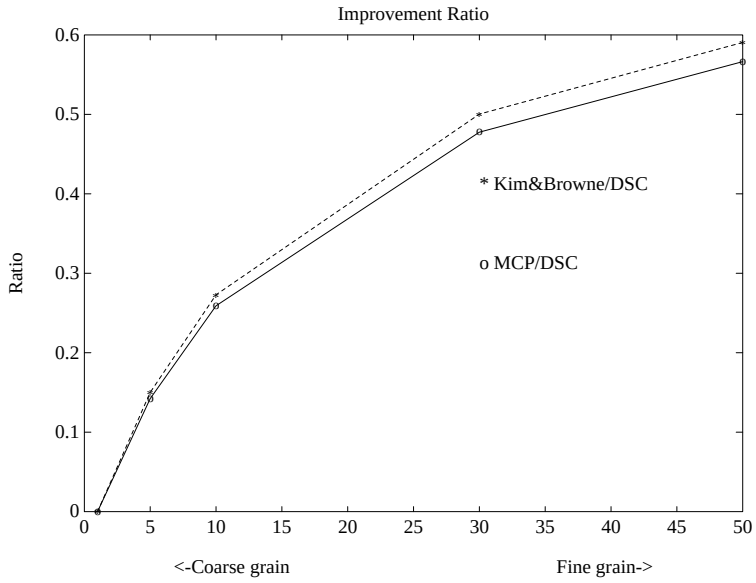


Figure 4.12: Scheduling the Choleski decomposition DAG.

When $1 \leq 1/g \leq 5$, the PT improvement ratio is within 15%. The improvement becomes larger along with the increase of $1/g$. In terms of complexity performance, DSC costs much less than the other two algorithms.

4.7 Conclusions on clustering

We have demonstrated that DSC is a superior algorithm in terms of complexity and performance. Considering the importance of communication for the new scalable architectures, such as nCUBE-II and INTEL iPSC/860, DSC is useful in scheduling such architectures, especially after the new communication technology, such as wormhole communication, makes the aforementioned architectures look like fully connected architectures.

Chapter 5

Processor Assignment of Clusters and Task Execution Ordering

Having determined a clustering for a DAG, we need to assign them onto p physical processors and order the task execution within each processor.

In Section 5.1, we discuss algorithms and present some experimental results on the mapping of clusters onto processors.

Then we focus on the problem of task ordering. We first analyze the theoretical properties of the classical critical path algorithm that lead to its near optimal performance in practice when communication cost is ignored, then we extend this analytic result to the task ordering problem. Section 5.2 gives a brief induction on the ordering problem. Section 5.3 introduces a framework for list scheduling with zero communication. Section 5.4 proposes algorithms for ordering based with nonzero communication. Section 5.5 presents optimality and comparative results to verify our analysis.

5.1 Processor Assignment of Clusters

Sarkars proposed an incremental scheduling method to map clusters to p processors. Assume that the number of clusters is u.

> After u clusters are derived, map those clusters to p processors by using a priority list scheduling. Assume that v tasks are sorted in a descending order of their priorities. The tasks are scanned from left to right. The scanned node, along with the cluster that it belongs to, is mapped onto one of the p processors that results in *the minimum increase of parallel time*. The parallel time is determined by executing the scheduled clusters in the physical processors and the unscheduled clusters in virtual processors.

The cost of this algorithm is $O(pv(v+e))$ which is time-consuming for a large graph.

We use a simple algorithm of complexity $O(v \log v + p^3)$. It performs this mapping in two steps:

1. Merge u clusters into p clusters.
2. Map the p clusters into p physical processors so that high communication clusters are close to each other in the network.

5.1.1 Cluster merging

We use a variation to *work profiling method*, George et. al. [38] for cluster merging. This method is simple, with complexity $O(v \log v)$ and has been shown to work well in practice, e.g. Saad [87], Geist and Heath [37], Ortega [77], Gerasoulis and Nelken [39].

1. Compute the arithmetic load LM_j for each cluster. Determine the average load $A = \sum_{j=1}^{u} LM_j/p$.
2. Sort the clusters in an increasing order of their loads.
3. Assign virtual processors to those clusters with $LM_j \geq A$.
4. Use wrap or reflection mapping for the remaining clusters.

Assume after step 3, the remaining virtual processors are numbered as $0, 1, \cdots, q-1$, and the remaining clusters are $M_1, M_2, \cdots, M_r$ in an increasing order of their load. The virtual processor for cluster M_j where $1 \leq j \leq r$ is defined as following with wrap mapping:

$$VP(j) = (j-1) \bmod q.$$

The formula for reflection mapping can be found in Geist and Heath [37].

The main difference of our approach and the classical load balancing method in [38] is that in our case only the clusters with loads less than the average are merged. Clusters with higher than average load are mapped directly to separate processors. In this way, the load is better balanced. For the GJ example with the natural linear

clustering we have

$$LM_j = \sum_{i=1}^{j} n\omega = jn\omega, \qquad A = \frac{\sum_{j=1}^{n} LM_j}{p} = \frac{n^2(n+1)\omega}{2p}$$

implying that all clusters with

$$j \geq \frac{n(n+1)}{2p} = t$$

will be mapped in separate processors while for $j < t$ the wrap mapping is used. The classical method will apply the wrap mapping to all clusters $j = 1 : n$.

Next we examine the performance of the above simple heuristic compared with Sarkar's expensive algorithm using the same clustering derived by DSC. We randomly generate 100 DAGs and weights as follows: generate randomly the number of tasks and edges and then assign randomly computation and communication weights. The size of the graphs varies from a minimum average of 143 nodes and 264 edges to a maximum average of 354 nodes and 2620 edges. In our experiments, the number of processors is chosen based on the widths of the graphs. The *width* and *depth* of graphs vary from 8 to 20 and thus we choose $p = 2, 4, 8$. Also to see the performance for both fine and coarse grain graphs we vary the granularity by varying the ratio of average computation over communication weights from 0.1 to 10.

Figure 5.1 shows that the average improvement ratio:

$$1 - T(\text{Sarkar})/T(\text{Load balancing})$$

where T() is the parallel time. The ratio of Sarkar's algorithm vs. the load balancing heuristic is between 10% to 35%. When the width of the graph is small compared to the number of processors, e.g. $p = 8$, Sarkar's algorithm is better than load balancing by about 30%. On the other hand, when the width is much larger than the number of processors then the performance differences are getting smaller, especially for coarse grain graphs, e.g. for $p = 2$ the improvement ratio reduces to about 10 % for coarse grain graphs. Intuitively, this is expected since each processor is assigned a larger number of tasks when the width to processor ratio increases and the ordering heuristic can better overlap the computation and communication.

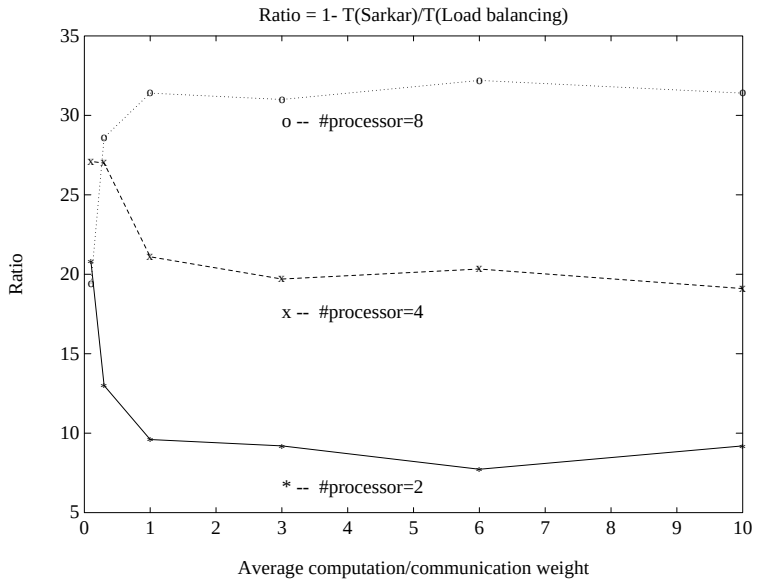


Figure 5.1: The performance of Sarkar's merging algorithm vs. load balancing algorithm. The graph width and depth are between 8 and 20.

With respect to the execution time of the heuristics in a Sun Sparcstation, the load balancing heuristic takes about 0.1 seconds to produce a solution for graphs with average $v = 200$ and $e = 400$, while Sarkar's algorithm takes about 40 seconds. When we double the graph size, the load balancing heuristic takes 0.2 seconds while Sarkar's needs 160 seconds. For the above graphs and p, the time spent for each graph varied from 0.05 to 0.3 seconds for the load balancing heuristic and from 9.8 seconds to 725 seconds for Sarkar's. On the average, the load balancing heuristic was 1000 times faster than Sarkar's for those cases.

To verify our conclusions we increased the width of graphs from 8-20 to 30-40 but then reduced the depth of graphs between 5-8 to keep the number of tasks sufficiently small for the complexity of Sarkar's algorithm. The results are shown in Figure 5.2 and are consistent with our previous conclusions. The performance of Sarkar's algorithm becomes better as the number of processors increases from $p = 2$ to $p = 16$ but then the performance reverses for $p = 32$, as expected, since p approaches the width of the graph.

Our experiments show that on average the performance of the load balancing algorithm is within 75% of Sarkar's algorithm for those random graphs. This is very

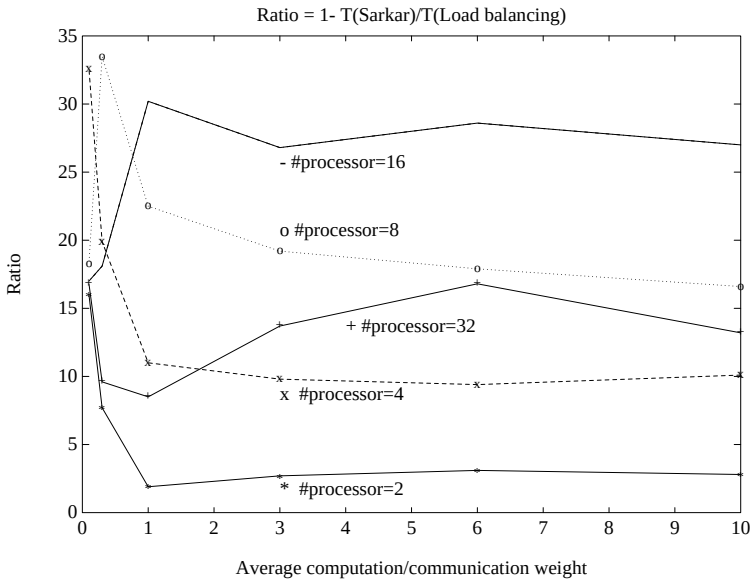


Figure 5.2: The performance of two merging algorithms for the graphs with width between 30 and 40 and depth between 5 and 8.

encouraging for the widely used load balancing heuristic.

5.1.2 Physical mapping

At this step, we have p virtual processors (or clusters) and p physical processors. Since physical processors are not completely connected, we have to take the processor distance into account. Let $TC_{i,j}$ be the total communication which is the summation of costs of all edges between virtual processor i and j. And Let $CC = \{TC_{i,j} | TC_{i,j} \neq 0\}$ and $m = |CC|$. In general we expect that $m << e$.

The goal of the physical mapping is to determine the physical processor number $P(V_i)$ for each virtual processor V_i that minimizes the following cost function F(CC).

$$F(CC) = \sum_{TC_{i,j} \in CC} distance(P(V_i), P(V_j)) \times TC_{i,j}.$$

Figure 5.3 is a special case of Gauss-Jordan DAG in Figure 3.8 when n=4. A clustering for this DAG is depicted in Figure 5.3(a) and each cluster corresponds a virtual processor if $p = 4$. Assuming that the edge cost is equal to 3 time units then the total communication between 4 virtual processors (clusters) is shown in Figure 5.3(b).

In Figure 5.3(c) we show one physical mapping to a 4-node hypercube with $F(CC) =$

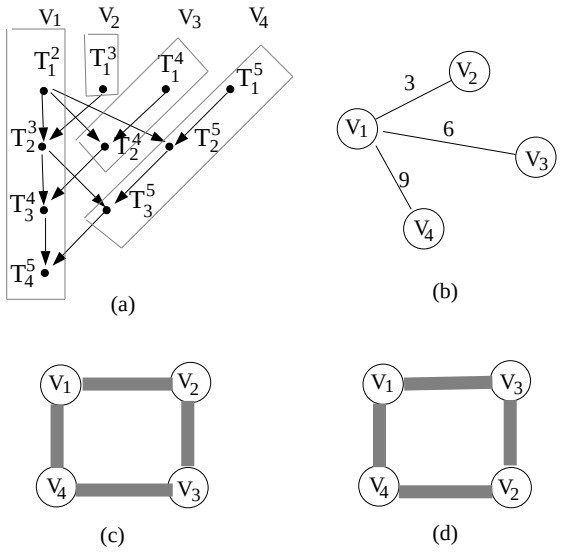


Figure 5.3: An example of physical mapping. Each nonzero edge cost is 3 time units.

24 and another mapping is show in (d) with $F(CC) = 21$.

Determining an optimal physical mapping is hard because this problem can be reduced to graph isomorphism problem. Currently we use a heuristic algorithm due to Bokhari [13]. This algorithm starts from an initial assignment, then performs a series of pairwise interchanges so that the $F(CC)$ is reduced monotonically. The complexity of this algorithm is $O(p^3)$. However we plan to implement different algorithms in future which have been studied extensively by Lo et.al [68].

5.2 Task Execution Ordering: An Introduction

Now we consider the following problem:

- Assume that a processor assignment for each task is given. The communication cost for tasks assigned in the same processor is zero but the communication between tasks in different processors is nonzero and is adjusted based the physical distance. Then determine an execution ordering of the tasks within each processor so that the parallel time is minimized.

The general task ordering problem above is NP-hard in stong sense [52]. Thus the main question is how to devise heuristics with good performance and low computational

complexity. This problem is closely related to the classical list scheduling. We will first analyze the theoretical properties of classical list scheduling algorithms and then use this analysis to propose heuristics that solve the above ordering problem.

In Section 5.3, we introduce a framework that considers list scheduling with zero communication as a series of processor mergings and study the theoretical properties of classical list scheduling algorithms. In Section 5.4, we address the task execution ordering problem. Here we have two choices of list scheduling. One is based on scheduling tasks from the free list and the other scheduling tasks from the ready[1] list. We introduce two such choices of ready list scheduling heuristics the RCP and RCP^*. Free list scheduling is not used since it creates idle gaps. In Section 5.5, we present two optimality results for RCP and RCP^*. We also present an experimental comparison for various choices. We have found that RCP^* has the best performance among those heuristics tested which is in agreement with our theoretical results.

The following definitions are used in this chapter:

- $L(x)$ – The level of a task n_x. This is the length of the longest path from n_x to an exit node, including nonzero communication delays in the path. The recursive definition is

  $$L(x) = \tau_x + \max_{n_j \in SUCC(n_x)} \{c_{x,j} + L(j)\}$$

  where $SUCC(n_x)$ is the set of all immediate successors of n_x.

- $L^*(x) = L(x) - \tau_x$, – The modified level excluding its execution time.

- $CPL = \max_{n_x \in V} L(x)$ –The length of the critical path of the given DAG.

- $\Delta T = \max_{n_j, n_i \in V} |\tau_j - \tau_i|$ –The maximum difference between the sizes of all tasks in the DAG.

- $ST(x)$ – The starting time mapping of task n_x in a schedule.

- $RT(x)$ – The ready time of task n_x. $RT(x) \leq ST(x)$.

[1] A task becomes *free* if all of its predecessors have finished their execution. A task becomes *ready* if it is free and all of the data that it needs from its predecessors have arrived locally to the processor that it has been assigned.

- $CT(x) = ST(x) + \tau_x$ – The completion time of task n_x.
- PT_A – The parallel time by scheduling algorithm A. PT_{opt} is the length of the optimal schedule.

We will need the results of the following Lemma.

Lemma 1

$$a - \max(c, d) \leq a - d$$

$$\max(a, b) - \max(c, d) \leq \max(a - c, b - d)$$

$$\max(a_1, ..., a_n) - \max(b_1, ..., b_n) \leq \max(a_1 - b_1, ..., a_n - b_n).$$

Proof: We have that

$$a - \max(c, d) = \begin{cases} a - d & \text{when } c \leq d \\ a - c \leq a - d & \text{when } c > d. \end{cases}$$

Using this inequality, we obtain

$$\max(a, b) - \max(c, d) = \max(a - \max(c, d), b - \max(c, d)) \leq \max(a - c, b - d).$$

It can be generalized further by induction as follows:

$$\max(a_1, ..., a_n) - \max(b_1, ..., b_n) \leq max(a_1 - b_1, \max(a_2, ..., a_n) - \max(b_2, ..., b_n))$$

$$\leq \max(a_1 - b_1, ..., a_n - b_n).$$

□

5.3 List Scheduling with Zero Communication

We first consider the problem of scheduling a DAG on p processors with zero communication overhead. The classical solution to this problem is priority list scheduling. Initially a list of tasks is ordered according to some given priority. At each scheduling step, the list scheduling heuristic schedules the highest priority free task on the first available processor. A generic procedure of list scheduling is in Figure 5.4.

What is interesting about list scheduling is that the performance is within 50% of the optimum independently of the priority list as shown by Graham [48]. An important

Prioritylist = $\{n_1, n_2, ..., n_v\}$ sorted in a descending order of task priorities;
clock = 0;
While Prioritylist is not empty **Do**
    Remove the left-most free task in the list and schedule it to an idle processor if available;
    clock = the next earliest time when a processor becomes available.
ENDWHILE

Figure 5.4: The algorithm of list scheduling.

question is what choices of priority lists will *consistently* give schedules that are "close" to the optimum schedule. Experimentally, Adam, Chandy and Dickson [1] answered the above question by conducting an extensive empirical performance study of CP algorithm with other four priority list scheduling algorithms. Their conclusion is that the CP heuristic is superior to others since it is near-optimal (i.e. within 5 percent of the optimal in 90 percent of random cases.) Theoretically, Coffman and Graham [23] show that CP is optimal for scheduling DAGs of equal task sizes on two processors. Also Hu [53] shows that CP is optimal for scheduling tree DAGs of equal task sizes on p processors.

Previous research has emphasized the worst case analysis in list scheduling. The experimental results in [1], however, raise an interesting question. Are there any underlying properties that characterize the near optimality of certain list scheduling algorithms such as CP? Furthermore the proof of optimality of CP for equal weight DAGs raises another interesting question. Why does CP perform so well when the tasks sizes are equal? We will investigate these two questions next.

5.3.1 List scheduling performance characteristics

To identify the important properties of list scheduling heuristics with good performance we need to introduce a mathematical framework. We consider list scheduling as a sequence of *merging* operations controlled by a *clock*. Initially every task is mapped into v virtual processors and there are p physical processors. At each step a virtual processor is merged with a physical processor and the schedule is derived for this step. In

the final step, the virtual architecture (DAG) is converted into a physical architecture of p processors. Assuming that PT^i is the parallel time of the new schedule at the completion of merging step i, then the merging sequence produces a sequence of parallel times $PT^0, PT^1, \ldots, PT^v$. The parallel time at each step is determined as follows:

- Initially, there are v virtual processors and p idle physical processors after the last scheduled task in that processor. The parallel time for executing the DAG is the length of the critical path, i.e. $PT^0 = CPL$.
- At step i, a task is selected and is scheduled onto a physical processor. The corresponding virtual processor is merged with this physical processor and the total number of virtual processors is decreased by one to $v - i$. The clock is reset to the next earliest time when a physical processor becomes available for the execution of another task and also there is a free task available. PT^i is determined by executing all scheduled tasks on p physical processors and executing unscheduled tasks on $v-i$ virtual processors. *Each virtual processor can only start execution of an unscheduled task after the current global clock time.*
- Finally after step v, the number of virtual processors reaches zero and a schedule for p processors is produced with $PT = PT^v$.

At the completion of step i and before step $i + 1$ begins, let *clock* be the current global clock time. Let $FLIST$ be the list of free tasks and $WLIST$ be the list of tasks that are being executed at time *clock*. By being executed we mean that the tasks have started but not completed their execution at time *clock*. Then the parallel time can be derived from the following expression:

$$PT^i = \max(T_{others}, clock + \max_{n_x \in FLIST} L(x))$$

where

$$T_{others} = \begin{cases} \max_{n_j \in WLIST}\{ST(j) + L(j)\} & WLIST \neq \emptyset \\ 0 & otherwise. \end{cases}$$

All heuristic algorithms studied in [1] can be considered in this framework. We assume non-backtracking to avoid the high complexity, then the values of $PT^0, PT^1, \cdots,$

PT^v are monotonically increasing. *A greedy approach would be to have the minimum increase in the parallel time from one step to the next.*

Definition. Let PT^i_{lopt} be the optimum parallel time at step i, which is the solution of $\min\{PT^i - PT^{i-1}\}$ where the minimum is taken over all possible choices of free tasks in $FLIST$ at step i. Then a heuristic is called δ-*lopt* if $\max_i\{PT^i - PT^i_{lopt}\} \leq \delta$ where δ is a given constant. If $\delta = 0$ then δ-*lopt* heuristic is called *local optimum.*

The important question is, of course, if local optimality implies near-optimality in the sense of Adam, Chandy and Dickson [1]. While it is difficult to prove such result theoretically, we do know that the CP heuristic is near optimal [1]. Therefore it would be of interest to see if CP is local optimum. We expect local optimum heuristics to perform better on average than heuristics with large nonzero δ. We present an example to demonstrate this point.

In Figure 5.5, we show results of two list scheduling algorithms studied in [1]: the CP and HLFNET (highest levels first with no estimated computation times). The priority list for CP is $\{n_3, n_1, n_2, n_4, n_5\}$ since $L(3) = 6 \geq L(1) = 4 \geq \ldots \geq L(5) = 1$. On the other hand the priority list for HLFNET is $\{n_1, n_2, n_4, n_3, n_5\}$ since it assumes that each task weight is equal to unit time.

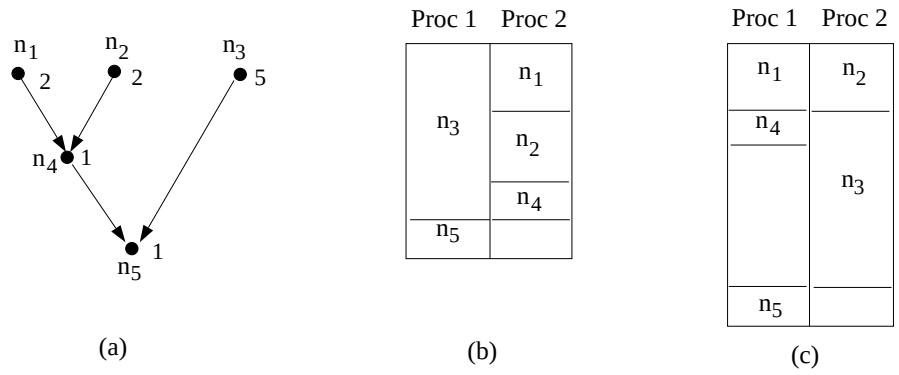


Figure 5.5: The impact of δ on performance of list scheduling. (a) is a DAG. (b) CP results in an optimal schedule with $PT = 6$ and $\delta = 0$ (c) $HLFNET$ results in $PT = 8$ and $\delta = 2$.

In Table 5.1, we show the merging sequences in CP list scheduling. For example, in step 1 n_3 is selected and scheduled in processor 1 because of its high priority. This choice is also the local optimum choice since scheduling either n_1 or n_2 instead will result in

the same parallel time. In Table 5.2, we show the HLFNET merging steps. In step 2, n_2 is scheduled on processor 2 but this is not the local optimum choice since n_3 scheduling in processor 2 results in shorter parallel time. To compute PT^2 in Table 5.2, we have $WLIST = \emptyset$, $FLIST = \{n_4, n_3\}$ at $clock = 2$ so that $PT^2 = clock + L(3) = 2 + 6 = 8$. Notice that $\delta_{CP} = 0$ while $\delta_{HLFNET} = 2$.

step i	selected	$FLIST$	$WLIST$	$clock$	PT^i	PT^i_{lopt}
0		$\{n_3, n_1, n_2\}$	$\{\}$	0	6	6
1	n_3	$\{n_1, n_2\}$	$\{n_3\}$	0	6	6
2	n_1	$\{n_2\}$	$\{n_3\}$	2	6	6
3	n_2	$\{n_4\}$	$\{n_3\}$	4	6	6
4	n_4	$\{n_5\}$	$\{\}$	5	6	6
5	n_5	$\{\}$	$\{n_5\}$	5	6	6

Table 5.1: Merging steps corresponding to CP scheduling in Figure 5.5(b).

step i	selected	$FLIST$	$WLIST$	$clock$	PT^i	PT^i_{lopt}
0		$\{n_1, n_2, n_3\}$	$\{\}$	0	6	6
1	n_1	$\{n_2, n_3\}$	$\{n_1\}$	0	6	6
2	n_2	$\{n_4, n_3\}$	$\{\}$	2	8	6
3	n_4	$\{n_3\}$	$\{n_4\}$	2	8	8
4	n_3	$\{n_5\}$	$\{n_3\}$	3	8	8
5	n_5	$\{\}$	$\{n_5\}$	7	8	8

Table 5.2: Merging steps corresponding to HLFNET scheduling in Figure 5.5(c).

The following theorem gives an upper bound for δ for the CP heuristic.

Theorem 5.1 *The CP heuristic is δ-lopt with $\delta \leq \Delta T = \max_{n_i, n_j \in V} |\tau_i - \tau_j|$.*

Proof: At step i, a task n_y with the highest level among free tasks is selected by CP and the corresponding virtual processor is merged with an idle physical processor, so that

$$PT^i_{CP} = \max(T_{others}, clock + \max_{n_x \in FLIST} L(x)) = \max(T_{others}, T_{free}(y))$$

where

$$T_{free}(y) = \max(clock + L(y), clock + \Delta clock_y + \max_{n_x \in FLIST - \{n_y\}} L(x))$$

$$\Delta clock_y = \text{the next processor available time} - clock.$$

Let task n_w be the optimal choice that results in PT^i_{lopt}. Using Lemma 1,

$$\begin{aligned} PT^i_{CP} - PT^i_{lopt} &= \max(T_{others}, T_{free}(y)) - \max(T_{others}, T_{free}(w)) \\ &\leq \max(0, T_{free}(y) - T_{free}(w)). \end{aligned}$$

Let $D = T_{free}(y) - T_{free}(w) = \max(H_y, S_y) - \max(H_w, S_w)$ where

$$H_z = clock + L(z), \quad S_z = clock + \Delta clock_z + \max_{n_x \in FLIST - \{n_z\}} L(x).$$

Note that $n_w, n_y \in FLIST$, and $L(y)$ is the largest, so that

$$T_{free}(w) = S_w = clock + \Delta clock_w + L(y), \text{ and } \quad S_y \geq H_w, \quad S_w \geq H_y$$

implying $D = \max(H_y, S_y) - S_w$.

If $D \leq 0$, then $PT^i_{CP} - PT^i_{lopt} \leq \max(0, 0) = 0$.

If $D > 0$, then $S_y \geq H_y$, because $S_w \geq H_y$. Thus

$$D = S_y - S_w = \Delta clock_y - \Delta clock_w + \max_{n_x \in FLIST - \{n_y\}} L(x) - L(y).$$

Since $|\Delta clock_y - \Delta clock_w| \leq \Delta T$ and $L(y)$ is the largest, $PT^i_{CP} - PT^i_{lopt} \leq \Delta T$. □

The following corollary is a direct result of the last theorem.

Corollary 5.1 *CP scheduling is local optimum for DAGs with equal weights.*

Local optimality, of course, does not imply optimality but we expect a near optimum performance of heuristics that possess this property. Adam, Chandy and Dickson [1] have provided the experimental evidence for such a performance for a local optimum heuristic. Furthermore for a tree DAG with equal weights CP scheduling is optimum on arbitrary number of processors Hu [53] and for any DAG with equal weights CP is optimum on two processors Coffman and Graham [23].

5.4 Execution Ordering with Nonzero Communication

In this section, we extend the previous analysis to the task execution problem:

- Given p clusters of unordered tasks mapped onto p physical processors. Determine a schedule for these clusters.

Figure 5.6 shows p clusters of unordered tasks (virtual processors) and p physical processors. Because the physical distance and inter-cluster communication delay are known in advance of execution, the level information $L(x)$ that includes inter-processor communication cost is deterministic. We can therefore extend the list scheduling to this problem as follows:

- First assign a priority value to each task. For example, the CP priority can be assigned by using the level information $L(x)$ that includes both the task computation and edge communication in the longest path.

- Maintain a local *free* task list and a local clock for each processor. The local clock time $lc(j)$, $j = 1 : p$ is set to the maximum between the earliest time that the processor is available for execution and the earliest time that the local free list becomes nonempty. The global clock time is defined as

$$gclock = \min_{j=1:p} lc(j) \text{ and processor } j \text{ has at least one free task.}$$

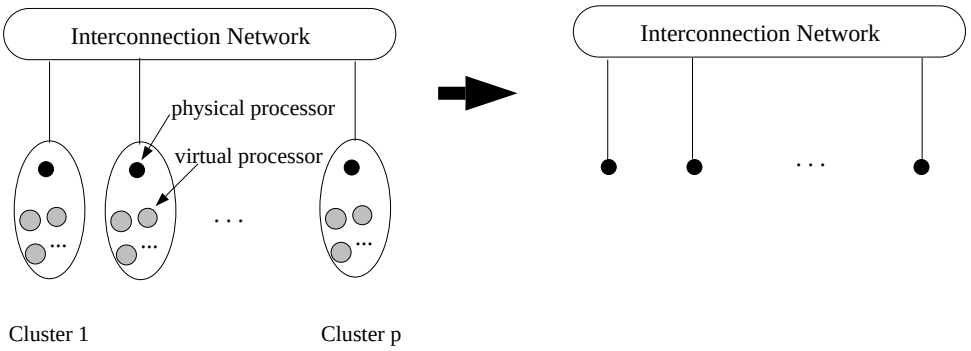
- For each scheduling step the processor that is available at time *gclock* and has the highest priority free task is chosen and that task is scheduled. In the case of a tie in priorities then the processor with the smallest processor number in $1 : p$ is chosen, i.e. in alphabetical order.

When communication delay is zero, a task selected by the classical list scheduling algorithm is ready for execution as soon as it becomes free. In the presence of communication, however, this is no longer true. If a free, but not-ready, task is scheduled it must wait for the data to arrive before it can start its execution and an idle gap is created in the schedule. On the other hand, if a ready task is chosen for scheduling then this task can start its execution immediately. So in the presence of communication we have two choices of list scheduling:

- Free list scheduling (FLS). Each processor maintains a free list of tasks and the highest priority free task is scheduled at the time that this processor becomes available.

- Ready list scheduling (RLS). Each processor maintains a ready list of tasks instead of free list and the highest priority ready task is scheduled at the time that this processor becomes available.

We call the free list scheduling with the CP priority as FCP(Free Critical Path), the ready list scheduling with the CP priority as RCP (Ready Critical Path). The ready list scheduling step is the same as the free list scheduling step with the exception that we replace *ready* list instead of *free* list in each step above.



(a) Ideal cluster architecture (b) Physical architecture

Figure 5.6: Transformation from a virtual architecture to the physical architecture.

Following the model presented in section 5.3.1, we can think of the above list scheduling as a sequence of virtual to physical processor merging operations depicted in Figure 5.6. The parallel time computation PT^i is similar to the computation described in the previous section. The main difference is that here each physical processor initially owns a set of virtual processors corresponding to the tasks pre-assigned to that physical processor, and it maintains a local clock and a free task list at each step. *Each virtual processor at each step now can start execution of a task only after the time of the local clock for this processor.*

The parallel time can be computed as follows. Assume that before starting step $i+1$, that is at the completion of step i, $FLIST(j)$ and $RLIST(j)$ are the local free

and ready task lists of processor j, for $1 \leq j \leq p$. Then

$$PT^i = \max_{1 \leq j \leq p} \{\max(F(j), W(j))\}$$

where

$$F(j) = \begin{cases} lc(j) & FLIST(j) = \emptyset \\ \max_{n_k \in FLIST(j)} \{\max(lc(j), RT(k)) + L(k)\} & otherwise \end{cases}$$

$$W(j) = \begin{cases} ST(x) + L(x) & n_x \text{ is the last task executed before time } lc(j) \text{ in Proc } j \\ 0 & \text{no such task exists.} \end{cases}$$

We give an example for clarification.

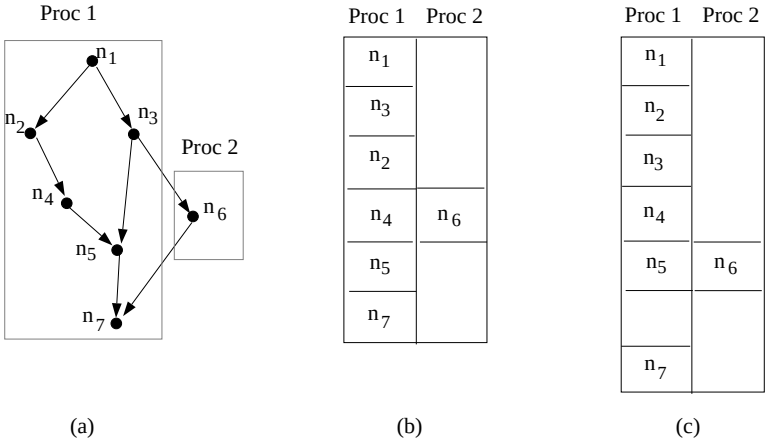


Figure 5.7: (a) A processor assignment of a DAG with unit computation and communication weights. (b) A schedule by FCP and RCP scheduling (c) Another schedule based on priority order $\{n_1, n_2, n_3, n_4, n_6, n_5, n_7\}$.

Figure 5.7(a) shows a DAG with unit computation and communication weights. Without a processor assignment given, the levels of tasks n_2 and n_3 are $L(2) = 7$ and $L(3) = 5$. When a processor assignment is given, as shown in Figure 5.7(a), then the edges within a processor are zeroed and $L(2) = 4$ and $L(3) = 5$. The schedule shown Figure 5.7(b) is derived by RCP or FCP. The CP priority order is $\{n_1, n_3, n_2, n_4, n_6, n_5, n_7\}$ and a detailed description of each merging step is shown in Table 5.3 for FCP and Table 5.4 for RCP. The only difference between FCP and RCP for this example is the task selection at step 4 derived as follows: At the completion of step 3 in Table 5.3, $lc(1) = 3$ and $lc(2) = 2$, and the global time is set to

$gclock = 2$. Processor 2 is available at time 2 but processor 1 is not since it executes n_2 at that time. Thus n_6 is picked up for scheduling and $lc(2) = 2+2 = 4$ where the first 2 is the old clock time and the second 2 is due to both communication and computation delay for this task. For Table 5.4 step 4 the differences is in $gclock = 3$ because of $lc(1) = 3$ and $lc(2) = 3$ at the completion of step 3. Thus at time 3 both n_4 and n_6 can be chosen. We choose n_4 for scheduling because of the alphabetical tie breaking rule.

step i	$gclock$	Selected	$FLIST(1)$	$lc(1)$	$FLIST(2)$	$lc(2)$	PT^i
0			$\{n_1\}$	0	$\{\}$	0	6
1	0	n_1	$\{n_3, n_2\}$	1	$\{\}$	0	6
2	1	n_3	$\{n_2\}$	2	$\{n_6\}$	2	6
3	2	n_2	$\{n_4\}$	3	$\{n_6\}$	2	6
4	2	n_6	$\{n_4\}$	3	$\{\}$	4	6
5	3	n_4	$\{n_5\}$	4	$\{\}$	4	6
6	4	n_5	$\{n_7\}$	5	$\{\}$	4	6
7	5	n_7	$\{\}$	6	$\{\}$	4	6

Table 5.3: FCP scheduling corresponding to Figure 5.7(b).

For the computation of the parallel time at the completion of step 4 of FCP table 5.3, we have

$$F(1) = \max(lc(1), RT(4)) + L(4) = \max(3,\ 3) + 3 = 6, \qquad F(2) = lc(2) = 4$$

$$W(1) = ST(2) + L(2) = 2 + 4 = 6, \qquad W(2) = ST(6) + L(6) = 6$$

which imply that $PT^4 = 6$.

The schedule in Figure 5.7(c) is based on the priority order $\{n_1, n_2, n_3, n_4, n_6, n_5, n_7\}$. This priority order is derived by using the task level values and assuming that all edges have nonzero communication even when they are within a processor. After the execution of n_1, the tasks n_2 and n_3 are both free and ready. Since n_2 has higher priority than n_3, n_2 is scheduled first. Since n_3 has been preassigned in processor 1, it has to be scheduled after the completion of n_2 even it is ready and processor 2 is idle. As a result, the ready time of n_6 is delayed. The overall parallel time is 1 time unit longer than that of Figure 5.7(b). This example also demonstrates the importance of accurate level information which is difficult to compute if the clusters to processor assignment is not known in advance.

step i	$gclock$	Selected	$RLIST(1)$	$lc(1)$	$RLIST(2)$	$lc(2)$	PT^i
0			$\{n_1\}$	0	$\{\}$	0	6
1	0	n_1	$\{n_3, n_2\}$	1	$\{\}$	0	6
2	1	n_3	$\{n_2\}$	2	$\{n_6\}$	3	6
3	2	n_2	$\{n_4\}$	3	$\{n_6\}$	3	6
4	3	n_4	$\{n_5\}$	4	$\{n_6\}$	3	6
5	3	n_6	$\{n_5\}$	4	$\{\}$	4	6
6	4	n_5	$\{n_7\}$	5	$\{\}$	4	6
7	5	n_7	$\{\}$	6	$\{\}$	4	6

Table 5.4: RCP scheduling corresponding to Figure 5.7(b).

5.4.1 *δ-lopt* analysis for ready list scheduling

As we mentioned earlier, the problem with FLS is that it creates an idle gap when scheduling a free task on a processor. In this section we only present a *δ-lopt* analysis for a ready list scheduling heuristics . We will compare the performance of FLS and RLS in section 5.5.2.

We consider two RLS heuristics: the RCP based on the highest level $L(x)$ as mentioned at last subsection and the RCP^* based on $L^*(x)$ priorities. Both of these heuristics can be implemented in $O(v \log v + e)$ time. For tie breaking between tasks we can use the MISF principle.

The *δ-lopt* definition here is slightly different that one given for list scheduling with zero communication. *The local optimum parallel time* PT^i_{lopt} *is derived over all possible schedules of ready tasks in* $RLIST(j)$ *for a selected processor* j. Then the following theorems are true.

Theorem 5.2 *The RCP heuristic is δ-lopt with* $\delta \leq \Delta T$.

Proof: We need to prove that $PT^i_{RCP} - PT^i_{lopt} \leq \Delta T$.

Assume at time $lc(u)$ of step i, ready task n_x at cluster u is selected for execution on physical processor u. Then the next available time for processor u at step $i+1$ is $lc(u) + \tau_x$. Thus

$$PT^i = \max_{1 \leq j \leq p} \{\max(F(j), W(j))\} = \max(T_u(x), T_{others})$$

where $T_{others} = \max_{j \neq u}\{\max(F(j), W(j))\}$. And

$$T_u(x) = \max(F(u), W(u)) = \max(H_x, \quad S_{z,x} | n_z \in FLIST(u) - \{n_x\})$$

where H_x is the length of longest path that goes through n_x:

$$H_x = \max(lc(u), RT(x)) + L(x) = lc(u) + L(x)$$

and $S_{z,x}$ is the longest path that goes through free tasks other than n_x in processor u:

$$S_{z,x} = \max(lc(u) + \tau_x, RT(z)) + L(z).$$

Now assume that n_y is the task with the highest level among ready tasks selected by RCP and ready task n_w is the optimal choice for processor u that minimizes $PT^i - PT^{i-1}$ over all ready tasks in $RLIST(j)$. Then according to Lemma 1,

$$\begin{aligned} PT^i_{RCP} - PT^i_{lopt} &= \max(T_u(y), T_{others}) - \max(T_u(w), T_{others}) \\ &\leq \max(T_u(y) - T_u(w), 0) \quad \leq T_u(y) - T_u(w) \\ &\leq \max(D, \quad S_{z,y} - S_{z,w} | n_z \in FLIST(u) - \{n_y, n_w\}) \end{aligned}$$

where $D = \max(H_y, S_{w,y}) - \max(H_w, S_{y,w})$.

For task $n_z \in FLIST(u), \quad z \neq w, z \neq y$, using Lemma 1:

$$\begin{aligned} S_{z,y} - S_{z,w} &= \max(lc(u) + \tau_y, RT(z)) + L(z) - \max(lc(u) + \tau_w, RT(z)) - L(z) \\ &\leq \max(\tau_y - \tau_w, 0) \quad \leq \Delta T. \end{aligned}$$

If $D \leq 0$, then $PT^i_{RCP} - PT^i_{lopt} \leq \max(0, \Delta T) \leq \Delta T$.

If $D > 0$, since $S_{y,w} \geq H_y, \quad S_{w,y} \geq H_w$, then $S_{w,y} \geq S_{y,w} \geq H_y, \quad S_{w,y} \geq H_w$.

Thus $L(w) \leq L(y)$ and $RT(w) \leq lc(u), \quad RT(y) \leq lc(u)$ imply:

$$\begin{aligned} D &= S_{w,y} - \max(H_w, S_{y,w}) \quad = S_{w,y} - S_{y,w} \\ &= \max(lc(u) + \tau_y, RT(w)) + L(w) - \max(lc(u) + \tau_w, RT(y)) - L(y) \\ &= \tau_y - \tau_w + L(w) - L(y) \quad \leq \Delta T \end{aligned}$$

Hence,

$$PT^i_{RCP} - PT^i_{lopt} \leq \max(D, S_{z,y} - S_{z,w} | n_z \in FLIST(u) - \{n_y, n_w\}) \leq \Delta T.$$

□

Theorem 5.3 *The RCP^* heuristic is δ-lopt with $\delta \leq \Delta T$.*

Proof: The proof uses similar analysis and definitions as in the previous theorem and we only need to show $PT^i_{RCP^*} - PT^i_{RCP} \leq 0$.

Assume that n_y is the task with the highest $L^*(y)$ among ready tasks and it is selected by algorithm RCP^*. Also assume that ready task n_w is selected by RCP and $n_y \neq n_w$. Thus $L^*(w) \leq L^*(y)$ but $L(w) \geq L(y)$ implying $\tau_w \geq \tau_y$.

Consider $PT^i_{RCP^*} - PT^i_{RCP} \leq \max(D, \ \ S_{z,y} - S_{z,w} | n_z \in FLIST(u) - \{n_y, n_w\})$.

For the second term in the above right-hand side, $S_{z,y} - S_{z,w} \leq \max(\tau_y - \tau_w, 0) \leq 0$.

Now we consider $D = \max(H_y, S_{w,y}) - \max(H_w, S_{y,w})$. If $D \leq 0$, then $PT^i_{RCP^*} - PT^i_{RCP} \leq \max(0,0) \leq 0$. If $D > 0$, by $L^*(w) \leq L^*(y)$ and several other facts used in the previous theorem we have

$$D = \tau_y - \tau_w + L(w) - L(y) = L^*(w) - L^*(y) \leq 0.$$

Hence, $PT^i_{RCP^*} - PT^i_{RCP} \leq \max(D, \ S_{z,y} - S_{z,w} | n_z \in FLIST(u) - \{n_y, n_w\}) \leq \max(0,0) = 0$. □

5.5 Theoretical and Experimental Comparisons

5.5.1 Optimality on fork and join DAGs

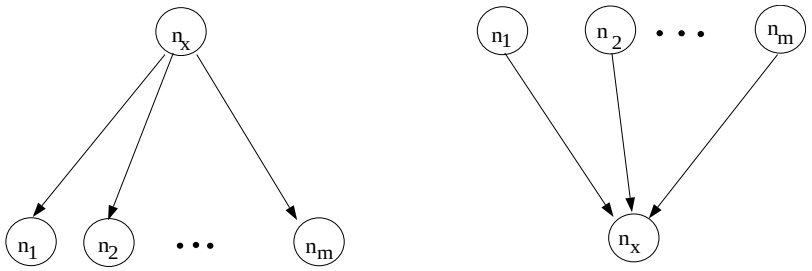
Finding the optimum in the general case is NP-hard in stong sense [52], even for chains of tasks. However, for a fork and join show in Figure 5.8. We show that they are solvable in a polynomial time by RCP and RCP^*.

Theorem 5.4 *Given a processor assignment for a fork DAG, RCP and RCP^* find an optimal schedule.*

Proof: For the fork in Figure 5.8(a), let the finishing time for each processor be FC, then

$$PT = \max_{1 \leq u \leq p} FC(u), \qquad FC(u) = \max_{PA(n_j)=u} CT(j).$$

We will show that both algorithms minimize FC for every processor and therefore find the optimum for this fork. We consider a cluster u for the following two cases.



(a) Fork (b) Join

Figure 5.8: Fork and join structures

Case 1: Root n_x is in u. Without loss of generality, we assume that other tasks in processor u are $n_1, n_2, ..., n_k$. Then the finishing time $FC(u)$ of this processor is always: $FC(u) = \tau_x + \sum_{1 \leq j \leq k} \tau_j$ which is the minimum.

Case 2: When n_x is not in processor u, assume that tasks assigned to u are $n_1, n_2, ..., n_k$. We prove the theorem in two parts. In part 1 we show that the length of the schedule derived by RCP or RCP^* (called schedule A) is the same as the one that executes tasks in increasing order of their ready times (called schedule B). In part 2, we show that schedule B is optimum.

Part 1 of case 2: We examine schedule A in u. When tasks become free after the execution of n_x, the leaf tasks in u may not be executed strictly in an increasing order of the task ready time. At time t, a ready task n_j with the highest $L(j)$ or $L^*(j)$ is executed and another ready task with earlier ready time could still remain in the ready list. Let us look at the sequence of task execution in processor u of schedule A. There could be some processor idle gaps in this schedule for tasks waiting for messages from n_x. Suppose tasks in this sequence are separated by idle gaps into groups $G_1, G_2, \cdots, G_s$. The ready time of tasks in any group G_j where $2 \leq j \leq s$ should be greater than that of tasks in group G_i for $1 \leq i \leq j-1$. For tasks within the same group G_j, we can always swap tasks in that group such that tasks are executed in an increasing order of their ready times, but the length of total execution time remains the same. Thus we can transform schedule A to schedule B by performing such a swapping for every group and the length of schedule A is equal to that of schedule B.

Part 2 of case 2: We now prove that schedule B is optimal by induction. Without loss of generality, we can assume that task execution order in processor u of schedule B is $n_1, n_2, \cdots, n_k$, i.e., $RT(n_1) \leq RT(n_2) \leq \cdots RT(n_k)$. Let $FC_k(u)$ be $FC(u)$ by schedule B when processor u has assigned k tasks. When $k = 1$, it is trivial. Assume $FC_k(u)$ is minimum for $k = s$.

When $k = s+1$, task n_{s+1} has the largest ready time and is scheduled as the last task. Then from the induction hypothesis, the execution time $FC_t(u)$ for $n_1, n_2, \cdots, n_s$ is minimum. The finishing time of processor u with task n_{s+1} is $FC_{s+1}(u) = \max(FC_s(u), RT(s+1)) + \tau_{s+1}$.

Suppose there is another schedule B' having less parallel time $FC'_{s+1}(u)$ when executing these $s+1$ tasks. We examine the execution order of tasks in schedule B'. Notice that the ready times for tasks assigned to processor u are fixed: $RT(j) = \tau_x + c_{x,j}$ for $1 \leq j \leq s+1$.

If n_{s+1} is not executed as the last task in B', since it has the largest ready time, all tasks executed after that task have less or equal ready times. Thus there is no idle gap between them. Then we can always swap those tasks and make task n_{s+1} be the last task while the parallel time remains the same. We call the new schedule by such swapping B''. Thus the length of B'' is equal to that of B', and $FC''_{s+1}(u) = FC'_{s+1}(u) < FC_{s+1}(u)$.

Since $FC''_{s+1}(u) = \max(FC''_s(u), RT(s+1)) + \tau_{s+1}$ and we know $FC_s(u)$ is the minimum, $FC''_{s+1}(u) \geq \max(FC_s(u), RT(s+1)) + \tau_{s+1} = FC_{s+1}(u)$, which is a contradiction. Thus the statement for part 2 is true for $k = s+1$. □

Theorem 5.5 *Given a processor assignment for a join DAG, RCP^* find an optimal schedule. RCP finds the optimum when task sizes are equal.*

Proof: For Figure 5.8(b), we first find an optimal schedule, then we show RCP^* derives the same schedule.

The optimal schedule S for any DAG G can always be reversed to be also optimal for the reversed DAG as follows:

1. Reverse the edge directions of G, the reversed DAG is called G^r.

2. Reverse the execution order of tasks in each processor in schedule S, the new schedule is called G^r. Then S^r is an optimal schedule for G^r.

Notice that a join in Figure 5.8(b) is the reversed DAG of a fork shown in (a). In part 2 of case 2 of the proof for Theorem 5.5.1, we have shown an optimal schedule B for a fork in Figure 5.8(a), which is to execute leaf tasks in each processor in an increasing order of the ready time $\tau_x + c_{x,j}$. Then the join, its reversed DAG in Figure 5.8(b), will have an optimal schedule B^r that executes leaf tasks in each processor in a descending order of $\tau_x + c_{x,j}$.

When RCP^* is used to schedule this join, all leaf tasks are ready for execution at the beginning and root n_x cannot be started unless all leaf tasks have been executed . Thus for each processor that contains leaf tasks, leaf tasks will be executed in a descending order of their priorities $L^*(j)$. Since $L^*(j) = c_{j,x} + \tau_x$ for each leaf task n_j and $c_{j,x}$ in the join is the same as $c_{x,j}$ in the fork, then that schedule is the same as B^r. Therefore, RCP^* derives the optimum for a join.

Algorithm RCP uses $L(j) = \tau_j + c_{j,x} + \tau_x$ as priorities. It will result in the same execution order if $\tau_j = \tau_i$ for $i \neq j$, i.e. leaf tasks have the same task execution times. □

5.5.2 Experiments with random graphs

In this section we present an experimental comparison of the seven heuristic algorithms:

1. RCP^* – ready task list scheduling with highest modified level first.
2. RCP – ready task list scheduling with highest level first.
3. $RCP0$ – it is the same as RCP except assuming edge communication cost as 0 in the level computation.
4. $RRND$ – ready task list scheduling with random numbers as task priorities.
5. $FRND$ – free task list scheduling with random numbers as task priorities, i.e. an algorithm that does not use any heuristic and only produces a valid schedule which satisfies the precedence constraints among tasks.

6. FCP – free task list scheduling with highest level first. It selects a highest level free task at each step. Since the selected task may not be ready at that time, an idle gap may be left.

7. FCP/F – it is the same as FCP but it will check whether there is a gap left. If there is, it tries to find some tasks to fill that gap. The algorithm is time consuming and its complexity exceeds $O(v^2)$. This idea is similar to the one used in the insertion heuristic algorithm ISH by Kruatrachue and Lewis [62].

We have used a program to generate random DAGs. The parameters that can be varied are the number of tasks v, the probability of having a precedence edge between two nodes, the task execution time and communication weights. The program first generates a number of task nodes within the given bounds randomly, and then generates edges between them randomly. Finally it assigns random numbers to node and edge weights.

To have a complete study of performance we vary the DAG parameters that may affect the performance of an algorithm. These are the size of the DAG, the sparsity of the edges (if $v < e < 3v$ then this is considered relatively sparse otherwise relatively dense), the ratio between the average communication and the average computation which is an estimate of the reciprocal of the granularity of the DAG, $1/g$, and the number of processors.

We generate the following 4 groups of DAGs with 25 random DAGs in each group and $1/g \approx 1$. SG1 and SG2 are relatively sparse and SG1 and DG1 are relatively small DAGs. The degree of parallelism, also known as the size of maximal independent task set or the width of the DAG, varies from 10 to 20.

1. SG1: DAGs where $70 < v < 210, 100 < e < 400$. Average, $v = 143$, $e = 264$.

2. SG2: DAGs where $230 < v < 500, 350 < e < 800$. Average, $v = 317$, $e = 609$

3. DG1: DAGs where $70 < v < 210, 400 < e < 1400$. Average, $v = 146$, $e = 810$.

4. DG2: DAGs where $230 < v < 500, 1000 < e < 5000$. Average, $v = 354$, $e = 2620$.

Given a DAG, we first fix the processor assignment. Then we run the above seven algorithms under the same processor assignment. We will use the improvement ratio to compare RCP^* with another algorithm, say, A:

$$Improvement = 1 - \frac{PT_{RCP^*}}{PT_A}.$$

We first report the results for $p = 4$ and $1/g \approx 1$. Afterwards we keep the same number of processors and vary the granularity. Finally we vary both the granularity and the number of processors.

	RCP	$RRND$	$RCP0$	FCP	FCP/F	$FRND$
SG1	0.1%	11.4%	2.2%	4.4%	2.0%	19.6%
SG2	0.3%	13.8%	1.0%	5.0%	2.2%	20.6%
DG1	1.7%	15.4%	3.5%	6.0%	4.2%	24.5%
DG2	0.5%	15.3%	1.9%	5.6%	3.0%	23.9%
Average	0.7%	14.0%	2.1%	5.3%	2.8%	22.2%
#cases RCP^*						
is better	33%,	100%,	62%,	100%,	95%,	100,%
same	62%,	0%,	35%,	0%,	5%,	0%,
worse	5%	0%	3%	0%	0%	0%

Table 5.5: The average improvement of RCP^* over other heuristics. $p = 4$ and $1/g \approx 1$.

	RCP	$RRND$	$RCP0$	FCP	FCP/F	$FRND$
SG1	-0.1%	15.5%	8.0%	14.2%	6.1%	34.8%
SG2	0.3%	15.9%	1.5%	12.1%	4.3%	39.4%
DG1	0.1%	11.5%	8.1%	11.6%	4.6%	36.7%
DG2	0.0%	17.6%	5.7%	13.5%	4.7%	41.7%
Average	0.1%	15.1%	5.8%	12.9%	5.0%	38.2%
#cases RCP^*						
is better	35%,	100%,	84%,	100 %,	97%,	100%
same	50%,	0%,	13%,	0%,	3%,	0%,
worse	15%	0%	3%	0%	0%	0%

Table 5.6: The average improvement of RCP^* over other heuristics. $p = 4$ and finer grain $1/g \approx 3$.

Table 5.5 shows the average improvement of RCP^* over other algorithms when the number of processors $p = 4$ and $1/g \approx 1$. In Table 5.6 we reduce the granularity by a factor of 3 so that $1/g \approx 3$, and in Table 5.7 we increase the granularity by a factor of 3 for every DAG so that $1/g \approx 0.3$, by increasing and reducing the communication

weights by a factor of 3.

	RCP	$RRND$	$RCP0$	FCP	FCP/F	$FRND$
SG1	0.0%	7.9%	0.7%	0.5%	0.4%	8.2%
SG2	0.1%	11.4%	0.5%	1.1%	0.7%	12.8%
DG1	2.5%	13.7%	2.8%	3.3%	3.0%	15.2%
DG2	0.4%	14.2%	1.3%	1.3%	1.1%	15.5%
Average	0.8%	11.8%	1.2%	1.6%	1.3%	12.9%
#cases RCP^* is better	35%,	48%,	100%,	90 %,	82%,	100%,
same	60%,	50%,	0%,	10%,	13%,	0%,
worse	5%	2%	0%	0%	5%	0%

Table 5.7: The average improvement of RCP^* over other heuristics. $p = 4$ and coarser grain $1/g \approx 0.3$.

From the above tables we see that for coarse grain DAGs the performance of heuristics based on critical path information is close. This is not true for finer grain DAGs in which case RCP^* outperforms all others. This is because RCP^* uses a much more accurate critical path information by including communication cost in the level computation. $RCP0$ and $RRND$ do not include communication weight in their task priorities. As expected from the previous theoretical analysis the performance of RCP^* and RCP is similar, with RCP^* slightly better on average.

Also notice that ready list scheduling heuristics perform better than the free list scheduling. This is because of the idle gap created by communication wait in free list scheduling. Increasing $1/g$ by enlarging the communication weight, free list scheduling will create bigger idle gaps and result in a worse performance. Even the FCP/F which tries to fill the idle gaps, with an additional computational cost, cannot perform better than RCP^* for most of cases.

An interesting question is if the above observed performance behavior is true when we also vary the number of processors. The width of those 100 random graphs varies from 8 to 20, thus we select the number of processors from 2 to 16. In Figure 5.9 we show the results for $p = 2, 4, 8, 16$ and $1/g \approx 0.3, 1, 3, 9$. The RCP performance is not shown because it is close to that of RCP^*. Also the average improvement ratio of RCP^* over $FRND$ is not shown because of the large variation from 5% to 50%

depending on the size of the granularity.

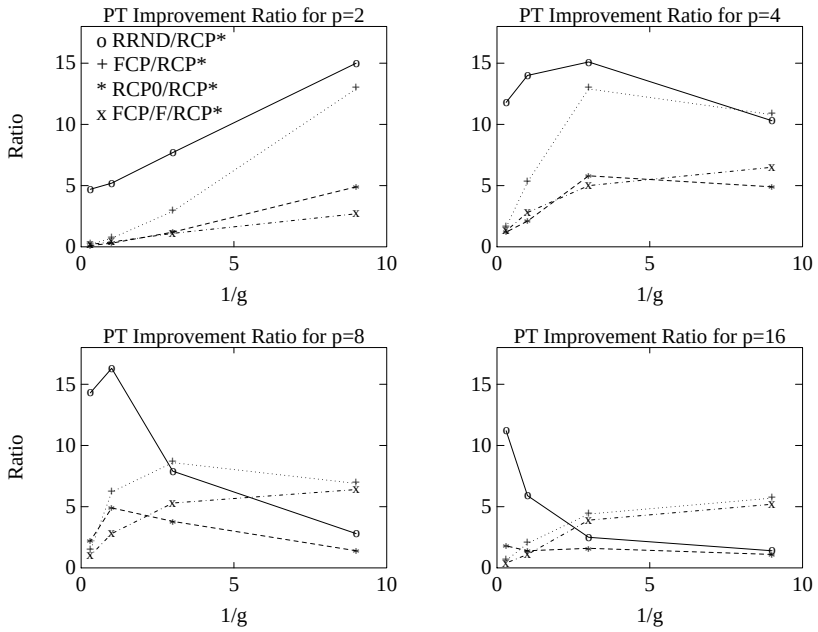


Figure 5.9: The improvement ratio when the number of processors varies.

Notice that the RCP^* is always better than the other heuristics. However, the improvement ratio does not always increase as graphs become finer grain (i.e. $1/g$ increases). There are many reasons for this phenomenon. The performance is affected not only by the granularity but also by the degree of the parallelism and the average number of free tasks or ready tasks available for selection at each scheduling step.

The following facts can make the performance of the heuristics similar.

- A1: By increasing the number of processors the number of ready (or free) tasks in the priority list is reduced at each scheduling step. Thus the selection choice is limited.

- A2: By decreasing the number of processors the number of ready tasks increases at each step and any heuristic has enough tasks to schedule.

- A3: By increasing communication the ready tasks are scattered in a much larger time period. Thus the number of ready tasks at each step decreases.

We now explain the results in Figure 5.9. For $p = 2$ the performance is the same as the one observed previously in Tables 5.5, 5.6, 5.7. As the graphs become finer grain the improvement of RCP^* increases. For the case $1/g \approx 0.3$ the improvement is not as high because the effect of A2 comes into play.

For $p = 4$ and $p = 8$, we see that the improvement increases up to a certain point and then it changes direction and starts to decrease. This is because of the influence of A3.

For $p = 16$, the improvement variation is less than 5%. The exception is the $RRND$ heuristic only in the extreme coarse grain case. This is explained by A1 and A3.

5.6 Conclusions on Execution Ordering

If the task graph weight information is not utilized for scheduling, the case of $FRND$, the performance could be more than 50% worse than RCP^*.

For coarse grain DAGs the difference in performance of heuristics that utilize accurate critical path information is insignificant provided that a good processor assignment is given.

For fine grain DAGs: (a) Ready list scheduling is superior to free list scheduling in general. (b) The variation of the performance for heuristics that use ready list scheduling and critical path information is within 15% on average. This result is similar to Adam, Chandy and Dickson [1] where they have observed a 10% variation between CP and random priority list when communication is zero.

Finally, a simple ready list scheduling heuristic with low complexity such as RCP^* will work extremely well for the task ordering problem.

Chapter 6

Code Generation

In order to execute tasks in a parallel machine, we need to generate the appropriate machine code using the previously-derived scheduling. In this chapter, we address the code generation problem for executing a general DAG on MIMD message passing machines. In Section 6.1, we discuss the machine model and the program semantics of DAG computation. We allow tasks in a DAG to access data in a shared memory programming style. In Section 6.2 we discuss a naive approach for code generation and further optimizations to improve the quality and performance of the code such as using aggregate communication primitives, eliminating unnecessary communication and improving memory utilization. In Section 6.3, we present a node program that incorporates optimization techniques without causing data inconsistency and deadlock. The node program correctly executes a DAG. In Section 6.4, we present a method for the iterative execution of a DAG, which incorporates a pipelining technique for executing a schedule between iterations. We also prove the correctness of the iterative execution of a DAG. In Section 6.5, we present an algorithm for implementing multicasting on hypercubes.

6.1 Computation Model

6.1.1 Machine model

The target machines are MIMD message passing architecture such as INTEL and nCUBE hypercubes with no global memory space. We assume that there are p processors available and each processor has a unique processor identification number and a local memory. Each processor can execute different programs on different data. Data communication and synchronization are implemented via message exchange. Several

asynchronous message passing primitives that will be used in this chapter are given below:

- *send(m,i)*

  A message with ID m is sent to processor i. Sending is non-blocking in the sense that the processor executing this procedure does not have to wait for an acknowledgment from the receiving processor.

- *multicast(m, D)*

  Message m is broadcasted asynchronously to multiple processor destinations defined by set D. This primitive is non-blocking. (This primitive may not be available in many machines. A multicasting algorithm on a hypercube will be given in the end of this chapter.)

- *receive(m)* and *probe(m)*

  Executing *receive*() will get a message with ID m if it is in the communication buffer of this processor. Otherwise, it waits idle until this message arrives. Function *probe(m)* checks if message m is in the local communication buffer.

- *receive*(m)*

  This is the same as receive() except that it will receive *any* message from the local communication buffer and assign the received message ID to m.

We number processors as $0, 1, 2, \cdots, p-1$. There is a special function *mynode*() that returns the processor number where the program resides. We assume that there is a special host node, named *host*, which performs the following functions: (1) Allocating the required number of processors (2) Loading the node programs, (3) Acting as a source task node in a DAG which sends the initial data to processors. We call this task as START. (4) Acting as a sink task which collects the final result from nodes. We call this task as END.

We make the following assumptions:

MA1: The communication network is reliable, i.e. no data messages are lost.

MA2: The host correctly delivers the initial data to the entry tasks of a given DAG and collects the final data sent from tasks.

6.1.2 The program semantics of DAG computation

From the point of view of a user, the task computation model can be described as follows: there exists a shared global memory containing a set of data items GM and there exists a set of tasks accessing data items in GM. Each task in a DAG reads some data items, performs a computation and writes some data items. The names of the data items and precedence edges between tasks play the role of synchronization.

We define the following sets for each task x in a DAG:

- $InMsg(x)$ – the data items that task x needs to reference for its computation. It is expressed by a set of data-to-source pairs: $InMsg(x) = \{(t_1, s_1), \cdots, (t_k, s_k)\}$ where each pair (t_i, s_i) indicates data t_i sent from task s_i. We have $t_i \in GM$ and $s_i \in V \bigcup \{START\}$.

- $OutMsg(x)$ – the data items produced by x that must be sent to the some successor tasks. It is expressed by a set of data-destination pairs: $OutMsg(x) = \{(t_1, s_1), \cdots, (t_k, s_k)\}$ where each pair (t_i, s_i) indicates data t_i sent to task s_i. We have $t_i \in GM$ and $s_i \in V \bigcup \{END\}$.

- $Procedure(x)$ – the operational specification of task x. It uses data items in $InMsg(x)$ and writes data items to $OutMsg(x)$.

The scheduling part of PYRROS determines the processor assignment for each task and task ordering within each processor. A scheduling result is specified by the following two functions:

- $PA(x)$ – the processor assignment of task x.

- $TaskExec(i)$ – the ordered set of tasks executed in processor i.

We assume that the scheduling result is *legal* in the sense that if we add pseudo-edges from any task x to y where x and y are mapped in the same processor and x is

executed before y, there should be no cycle with respect to both dependence edges and pseudo-edges.

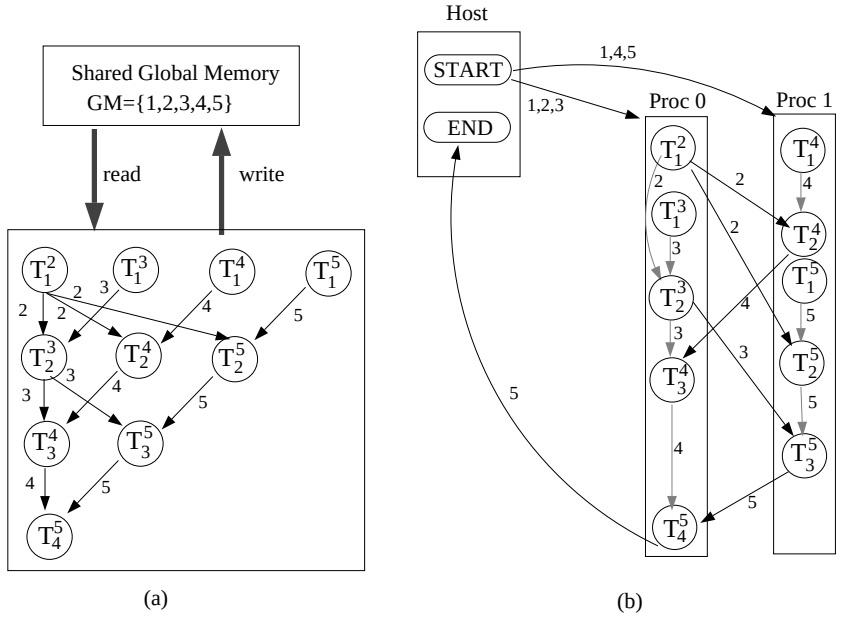


Figure 6.1: (a) The computation of the GJ DAG with $n = 4$ from the point of view of a user(b) The compiler's point of view after mapped to 2 processors.

Example: Figure 6.1(a) is the GJ DAG with $n = 4$. From the user's point of view, the shared global memory contains 5 columns and task T_k^j uses column k and j to redefine column j. Precedence edges are marked with the data items (columns) to be communicated. Thus $InMsg(T_k^j)$ contains column k and j while $OutMsg(T_k^j)$ contains column j. For example, $InMsg(T_1^4) = \{(1, START), (4, START)\}$ and $OutMsg(T_1^4) = \{(4, T_2^4)\}$. $InMsg(T_2^4) = \{(2, T_1^2), (4, T_1^4)\}$ and $OutMsg(T_2^4) = \{(4, T_3^4)\}$.

Figure 6.1(b) gives a legal mapping and ordering of this DAG onto 2 processors. The dashed arrows within each processor indicate the intra-processor communication. The task execution sets of two processors are: $TaskExec(0) = (T_1^2, T_1^3, T_2^3, T_3^4, T_4^5)$ and $TaskExec(1) = (T_1^4, T_2^4, T_1^5, T_2^5, T_3^5)$.

We now study the properties of a DAG. Since data accessing of tasks in a DAG must be synchronized through data dependence edges and there are three types of data dependencies between tasks (true, anti, and output) [95], we assume that a DAG satisfies the following conditions.

DA1: *A task receives distinct data items.*

No data items with the same ID are received from different tasks.

DA2: *True and anti data dependence.*

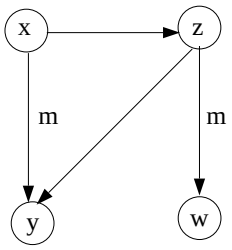
If task x produces data item m and another task y uses data item m, then there must be a path from x to y or from y to x.

DA3: *Output data dependence.*

If task x produces data item m, another task y also produces data item m, there must be a path from x to y or from y to x.

DA4: *A DAG is sequentialized.*

It is legal to execute tasks in this DAG sequentially with respect to a topological ordering. Namely a task writes a data item to GM and another task executed later based on that ordering should get the correct copy from GM. Figure 6.2 gives a counterexample. In Figure 6.2(b), y is not able to obtain the correct copy of data item m defined by x since it has been re-written by z. This DAG is not sequentializable. Theorems 6.1 and 6.2 further examine the conditions for satisfying DA4.



(a)

x writes data item m.
z writes data item m.
y reads data item m.
w reads data item m.

(b)

Figure 6.2: (a) A non-sequentializable DAG. (b) A sequential execution with incorrect result.

Figure 6.1(a) is such a DAG that satisfies those conditions. Since a dependence graph is usually derived from a sequential program, this graph is correctly executed in that sequential program which corresponds to *one* topological ordering. We need to see if this graph is sequentializable with respect to *any* topological ordering.

Theorem 6.1 *If a DAG G satisfies DA1, DA2, DA3 and it is sequentializable with respect to one topological ordering, then it is sequentializable with respect to any topological ordering.*

Proof: We prove it by contradiction. Let O_1 be a topological ordering which correctly executes G. Let another topological ordering O_2 be $\cdots t_i \cdots t_j \cdots t_k \cdots$ and assume that the sequential execution of G based on O_2 is not correct. In O_2, task t_k is supposed to read data m produced by t_i. However task t_j re-writes m and t_k is not able to get the copy of data m produced by t_i.

By DA3 there must be a dependence path from t_i to t_j. By DA2 there must be a dependence path from t_j to t_k. Then in the order O_1, t_i must be before t_j and t_j must be before t_k. Then the execution of tasks in G based on O_1 is also incorrect, which is a contradiction. □

Theorem 6.2 *Assume that a DAG G satisfies DA1, DA2, DA3. G satisfies DA4 iff for any two tasks x and z that define data item m and another task y that uses m produced by x, task z is not in a path from x to y.*

Proof: Assume that G satisfies DA4. If there exists such task z which is in a path from x to y, then in any sequential execution, task x is executed before z and y must be executed after z. Task y cannot read data item m produced by x, which is a contradiction.

Assume that G does not satisfy DA4 and there is an ordering $\cdots t_i \cdots t_j \cdots t_k \cdots$ which results in an incorrect sequential execution. Task t_i produces data m for t_k but t_j over-writes it. By DA2 and DA3, we can see that task t_j must be in a path from from t_i to t_k. □

We will call conditions DA1, DA2, DA3 and DA4 together as DA. The above theorems show that a DAG derived from a sequential program based on a full data dependence analysis can easily satisfy DA. Any DAG that does not satisfy DA2, DA3, or DA4 can be transformed to one that satisfies them by renaming data. Cytron et al. [27] discussed a method for renaming to remove anti and output dependencies. A special approach is to associate a data item with its producer name such that no same

data ID with different data content will be used in this DAG. In this way, this DAG will satisfy the above conditions. The disadvantage of this naming is that the space for the same data items produced by different tasks cannot be shared, which could result in poor space utilization. Since most of parallel machines are used for solving problems of large sizes, the efficient utilization of memory space is also an important issue for code generation.

In the following sections, we will study how to generate code for a processor that correctly executes a DAG according to the given schedule on a message passing machine if DA is satisfied.

6.2 A Naive Approach to Program Generation

The main function of code generation is to produce code for each parallel processor such that every task is correctly executed. This code needs to make sure that before executing a task, its predecessors have finished computation and all data items referenced by this task are available in the local memory. After the completion of the computation each new data item produced must be sent to the appropriate processors whose tasks need these data for further computation.

For each task x in $TaskExec(i)$ in terms of the execution order **DO**
  For each edge (m, y) in $InMsg(x)$ **DO**
    If $PA(y) \neq i$, check if task y completed its computation and issue $receive(m)$.
  EndFor
  Execute $procedure(x)$.
  For each edge (m, y) in $OutMsg(x)$ **DO**
    If $PA(y) \neq i$, notify the completion of x to successor y and issue $send(m, PA(y))$.
  EndFor
EndFor

Figure 6.3: Node program for processor i. A naive approach.

A naive approach for program generation is to use the code for each processor i as shown in Figure 6.3. This approach is a precise interpretation of the DAG computation model without incorporating any optimizations. We now address the drawbacks with

this naive approach and discuss ways to improve performance.

1. **Reduce space cost for storing dependence information**

   In the naive approach, checking dependence satisfaction requires that each processors knows the names of predecessors and successors of a task executed in this processor. The space overhead for storing such information could be large. To reduce such space cost, we can use the availability of data items needed by a task as a mechanism to check the satisfaction of task precedence. We do not need to incorporate task names in data names. We will show the feasibility of this approach when DA is true.

2. **Use aggregate communication**

   A task could send the same message to many other tasks in different processors. The naive approach repeatedly uses one-to-one sending, which could create communication contention in the processor network. Thus a broadcasting scheme should be used to take advantage of the network topology in order to alleviate network message traffic.

3. **Eliminate unnecessary communication**

   If two tasks are assigned in the same processor, there is no need for communication between them. The code in Figure 6.3 can eliminate such unnecessary communication. However, there are other redundant communication in the naive approach. For example, in Figure 6.1(b) two task T_2^4 and T_2^5 assigned in processor 1 receive the same data item 2 from task T_1^2 in processor 0. Processor 0 issues two sends with the same message to processor 1, and processor 1 issues two receives for task T_2^4 and T_2^5 respectively.

   An optimized approach should eliminate such redundant communication. From the receiver's point of view, the receiving operation for T_2^5 can be eliminated since after T_2^4 has received data item 2 from processor 0, data can be stored in the local memory. Task T_2^5 can just fetch data from the local memory. Thus a redundant receiving can be detected if the data item has already been in the local memory.

From the sender's point of view, task T_1^2 needs to only send one copy of data to processor 1.

4. **Preserve data coherence and improve space utilization**

Unnecessary receiving communication can be eliminated if a data item to be received is already in the local memory. However, there is a data coherence problem that must be addressed: the local memory may not keep the correct copy of a data item. A receiving of a new copy of a data item m could be mistakenly eliminated when an old copy of m is still in the local memory.

For example, in Figure 6.4, y uses item m defined by x and w uses item m defined by z, and y is executed before w. Then after the execution of task y, data item m is left in the local memory. Another task h in processor i that uses m can get that data item directly from the local memory. Before the execution of task w, the local memory still holds a copy of data m which is not what task w needs. To keep the data coherence, we need to get rid of data item m from the local memory before executing w then task w is able to issue a receiving to get the new copy of data m sent from task z.

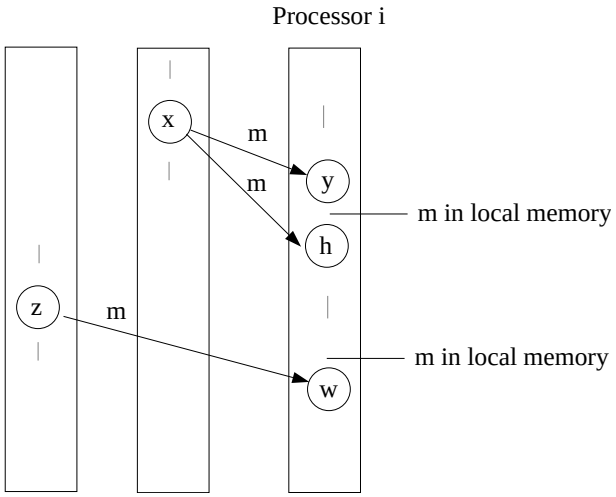


Figure 6.4: An example of data inconsistency.

Thus we need to maintain the following status of a data item m in the local memory: a data item is called *dead* at some time slot if no task in this processor will use the copy of this data item after that time. For Figure 6.4, the copy of

data item m produced by x will be dead before task w.

Deleting dead data items will not only preserve data coherence but also improve space utilization. Since the size of a local memory is still limited, we need to reuse the memory space. Thus if some data items are no longer used after certain execution point, their space should be released. For example, in Figure 6.1(b) data item 2 can be freed after the completion of T_2^5.

5. **Reduce communication idle time and avoid communication buffer overflow**

   In our asynchronous computation model, a data item may arrive at the local communication buffer of a processor earlier than the time it is used. For example, a task needs to receive data items m_1 and m_2 from some remote processors. The naive approach issues $receive(m_1)$ and then $receive(m_2)$. If m_2 has already arrived at the communication buffer but m_1 has not, the processor will remain idle until the arrival of m_1. Then it will fetch m_1 and m_2. Since the ordering of receiving is not important to the computation of this task, we can improve the efficiency by fetching m_2 from the communication buffer during the period that the processor is waiting for m_1. To do that, we need to use non-blocking message receiving $receive*()$ which will fetch any message from the buffer without knowing the message ID in advance. Notice the data item received using $receive*()$ may not be needed for immediate use. In this way, idle time is reduced and data items in the communication buffer can be moved to the local memory as early as possible. Moreover, it will also alleviate the problem of buffer overflow since the messages are read as soon as possible and as a result the buffer space is released.

In the next section, we are going to discuss how PYRROS incorporates those ideas to optimize performance.

6.3 An Optimized Approach for Executing a DAG

In Figure 6.5, we list the optimized code for processor i where $LM(i)$ is a set of data items in the local memory of this processor. In the receiving stage of task x, a receiving of a data item is eliminated if this item is in the local memory. If m is in the communication buffer, then *receive(m)* is issued otherwise *receive*()* is issued to fetch other messages in the buffer. This is repeated until m is in the local memory. Such setting will allow a data item to be removed from the communication buffer as soon as possible.

In the sending stage, produced data items are not sent one by one. The processor first accumulates and classifies all sending requests. Then for each data item, it uses multicasting if applicable otherwise uses one-to-one sending. After the sending stage, dead data items are released from the local memory to preserve data coherence and improve memory utilization.

Notice that each processor does not need to know the names of the predecessors of a task. The task precedence is satisfied automatically by checking the availability of data items used by a task. However, a processor may still need to know the successors of a task. Comparing to the naive approach, we save 50% in space overhead for storing dependence information.

In the next subsections, we will first give a formal study of the correctness of this code and then we present a method for the detection of dead data items.

6.3.1 Correctness analysis

We need to study the correctness of the code in Figure 6.5. A common error in programming message passing distributed memory machines is deadlock, e.g. Masticola and Ryder [71], Ladkin and Simons [64]. Because we allow shared memory programming style in a DAG, another difficulty arises when different tasks at different times modify the same data items. Thus we need to make sure each task receives the correct copies of data items.

Definition: The execution of node programs for a DAG is *correct* if

For each task x in $TaskExec(i)$ in terms of the execution order **DO**
  For each edge (m, y) in $InMsg(x)$ **DO**
    While $m \notin LM(i)$ **Do**
      Use $probe(m)$ to check if m is in the communication buffer. If yes, use *receive(m)* otherwise *receive*(n)*. Copy the received data item to $LM(i)$.
    EndWhile
  EndFor
  Execute $procedure(x)$.
  Send produced data items: if a destination is in processor i, directly put it in $LM(i)$. For other destinations, use $multicast()$ if applicable otherwise use $send()$.
  Delete those *dead* data items from $LM(i)$ and release their space.
ENDFOR

Figure 6.5: Optimized code for processor i.

1. Each task is executed and *all of its dependencies are satisfied*: it receives all data items specified by its incoming edges and sends data out to its successors.

2. The execution is *deadlock-free* if : (a) there is no outstanding receive such that a receiving is issued in processor i but no data item is sent to processor i. (b) There is no outstanding send such that a data item is sent to processor i, but no receiving is issued in processor i.

We need to use the following two lemmas in proving correctness.

Lemma 6.1 *The following case is impossible: task y defines data m and sends it to task x. Another z defines data m and sends to task w. Task y and z finish execution before x and w.*

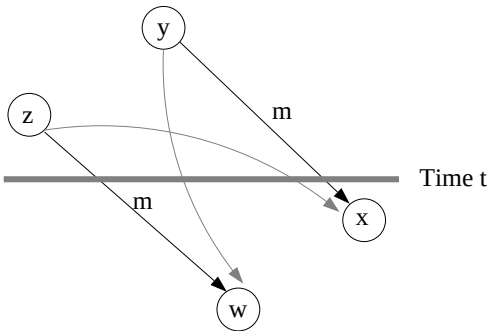


Figure 6.6: The dependence relations between tasks for Lemma 6.1.

Proof: If $x = w$, then task x would receive the same data from y and z, which is impossible from DA1. Thus $x \neq w$.

Now we prove the statement by contradiction. Assume the opposite is true and y and z finish their execution before x and w. According Assumption DA2, there is a dependence path from y to w and a path from z to x as shown in Figure 6.6.

We also know there is a path between y and z in terms of Assumption DA3. If this path is from y to z, then x uses m from y and z redefines m, which contradicts the assumption DA4. If it is from z to y, then w uses m from z and z redefines m, which contradicts to Assumption DA4. □

Lemma 6.2 *If there is a new definition of data item m produced at processor i by task y at time t, and this copy is going to be used by task x in processor j, then the old copy of m in the local memory of processor j is discarded (if there is a copy) before any receiving is issued at processor j after time t.*

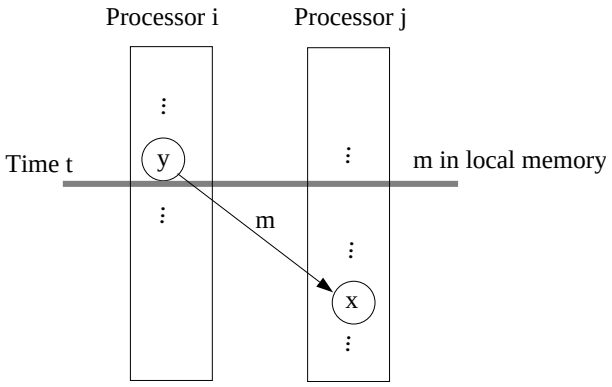


Figure 6.7: Scenario of Lemma 6.2.

Proof: The scenario for this lemma is depicted in Figure 6.7. Assume there is an old copy of data m in $LM(j)$. There are two cases:

- If $i = j$, the old copy of m in $LM(i)$ is replaced immediately at the completion time of task y (i.e., at time t) as shown in Figure 6.5.

- If $i \neq j$, assume the old copy is not discarded at processor j after time t. That indicates it is alive and some task w will use it in processor j. Assume task z originally produces that old copy. Thus we have a situation that z and y are executed before time t, and task x and w executed after time t, which is impossible according to Lemma 6.1. □

Theorem 6.3 *If DA for a DAG G is true, all tasks in G are executed and their dependencies are satisfied by the code in Figure 6.5.*

Proof: Within each processor, we add a pseudo-edge from task x to y if there is no dependence path from x to y and y is executed immediately after x. We prove this theorem by induction in terms of a topological task order on the new scheduled DAG.

1. By Assumption MA2, the START task is executed correctly. The first task at each processor either receives data from the START task or has no receiving. This task is executed and sends data to its successors.

2. Assume all predecessors of task x satisfy their data precedence and task x is assigned to processor i. Since tasks before x in processor i have been executed, we need to show that x receives the correct data for its execution.

Let task y be the predecessor of x that produces a data item m for x. After y has been executed, y must have sent data item m. We check if x receives that copy. There are two cases in the receiving stage of task x.

Case 1. $m \in LM(i)$.

- Subcase (a) $PA(y) = i$.

  After its execution, task y puts data item m in $LM(i)$ for x to read. There should be no task executed in processor i between task y and x that redefines m since it will violate Assumption DA4.

  Also there should be no other task executed in processor i between y and x that receives a copy of data m which is not defined by y. Suppose there exists such a task, say h, receiving m from z where $z \neq y$. Since the execution order in processor i is $\cdots y \cdots h \cdots x$, both z and y are executed before h and x, which is impossible by Lemma 6.1.

- Subcase (b) $PA(y) \neq i$.

  Task y at time t sends data m to processor i for task x or others to use. Then the old copy of m will be discarded before any receiving issued in processor i after time t according Lemma 6.2.

Now we claim that there is no other task h executed after t and before executing x that redefines m or receives a copy of m which is not defined by y.

- If there exists a task h that redefines m before the execution of x, then according to Assumption DA2 and DA3, there must be a precedence path from y to h and from h to x, which contradicts Assumption DA4.
- If there exists a task h that uses m which is not defined by y, say it is defined by z. We follow a similar proof as in Subcase (a) and exclude this scenario using Lemma 6.1.

Then the receiving of data m is eliminated and the code in Figure 6.5 gets the data copy directly from $LM(i)$.

Case 2. $m \notin LM(i)$.

Notice it is impossible that data item m produced by y has appeared in $LM(i)$ but has been already deleted as useless item before task x accesses it. Since if m had been received, that copy would not be dead as long as there is a task such as x accessing that copy.

Thus m is either in the communication buffer or on the way to that processor. By assumption MA1, network is reliable and eventually data m will arrive in the buffer. The *While* loop in Figure 6.5 makes sure that m is received eventually.

We claim it is impossible that the received m is not the one defined by y. Assume that it is defined by another z and is sent to task h, and h is executed later after x in processor i. Then y and z finish their execution before the execution of task x and h, which is impossible by Lemma 6.1.

Thus task x correctly receives data item m produced by task y. Then the execution proceeds successfully and $procedure(x)$ is executed and all of data items produced by x are sent out. □

From the proof of the above theorem, we can easily see:

Corollary 6.1 *A receiving of data item m in processor i is redundant if $m \in LM(i)$.*

Theorem 6.4 *The execution of the code in Figure 6.5 is deadlock free.*

Proof: By Theorem 6.3, all tasks are executed and the receiving procedures issued by each task successfully obtains data. Thus there is no outstanding receive.

Next we show there is no outstanding send. When a data item m is sent by y to processor i at time t, we assume that there are tasks $x_1 x_2 \cdots$ at processor i using m. Notice that y only issues one send of data m to processor i. Assume that x_1 is executed before x_2 and so on. By Theorem 6.3, x_1 must get that copy sent by task y. Other tasks x_2,$x_3 \cdots$ get the copy from $LM(i)$. If task y sends data to the host (the END node), the host correctly collects those data by Assumption MA2. □

6.3.2 The detection of dead data items

There are two approaches in determining if a data item is dead. One is based on static dataflow computation. But the result of this computation cannot be expressed uniformly and the generated code is in a MPMD (multiple program multiple data) style. Another is based on the run-time resolution which generates SPMD (single program multiple data) code.

Static dataflow computation: A MPMD approach

We can use dataflow computation to determine at each point of execution which data items become dead. For processor i, assume $TaskExec(i) = (n_1, n_2, \cdots, n_k)$. In Figure 6.8, we compute $MsgUsed(n_j)$ which is the set of data items that will be used in task $n_{j+1}, \cdots, n_k$. Operator $\ominus$ is defined as: $X \ominus Y = \{msg \mid msg \in X \wedge first(msg) \notin FirstSet(Y)\}$ where $first((n, s)) = n$, the first item of a pair. And $FirstSet(\{(t_1, s_1), \cdots, (t_k, s_k)\}) = \{t_1, t_2, \cdots, t_k\}$, the set composed of the first elements of the message items. Thus after the completion of task n_j, for any data item x in the local memory produced by task s, if (x, s) is not in $MsgUsed(n_j)$, then this data item is dead.

Example: For processor 1 in Figure 6.1(b), we perform the computation in Table 6.1. Item $(2, T_1^2)$ is always in $MsgUsed$ sets of T_1^5, T_2^4 and T_1^4 but is not in

$MsgUsed(n_k) = \emptyset$
FOR $j = k$ **TO** 2
  $MsgUsed(n_{j-1}) = (MsgUsed(n_j) \ominus (OutMsg(n_j) \cup InMsg(n_j))) \cup InMsg(n_j)$
ENDFOR

Figure 6.8: Static dataflow computation for dead data items.

$MsgUsed(T_2^5)$. Therefore data item 2 is dead after the completion of T_2^5. Similarly, item 4 is dead after the completion of T_2^4.

	$MsgUsed$
T_3^5	$\emptyset$
T_2^5	$\{(5, T_2^5), (3, T_2^3)\}$
T_1^5	$\{(5, T_1^5), (2, T_1^2), (3, T_2^3)\}$
T_2^4	$\{(5, START), (1, START), (2, T_1^2), (3, T_2^3)\}$
T_1^4	$\{(4, T_1^4), (5, START), (1, START), (2, T_1^2), (3, T_2^3)\}$

Table 6.1: An example of static data flow computation.

Since dead data items vary differently from time to time, it is difficult to express the result of static dataflow computation uniformly, hence MPMD code is generated. The disadvantage of such a MPMD code is that the code content depends on a specific static graph and the code size is proportional to the size of this graph.

Run time resolution: A SPMD approach

In order to generate SPMD code, which is problem size independent, we will use a run-time resolution method to detect dead data items. The time and space cost of such run-time method is $O(1)$, which is small.

At each processor i, we maintain a counter for each data item m in $LM(i)$: $usage(m)$.

- Assume that data m is stored in $LM(i)$ at time t. Then $usage(m)$ is the number of tasks executed at processor i after time t that use item m in their executions.
- We decrease $usage(m)$ by 1 at each reference to data item m by a task x in processor i after time t,
- A data item m in $LM(i)$ is dead if $usage(m) = 0$.

Notice that if task x uses m but also produces m, the *usage* of the old copy of m must be zero and $usage(m)$ is reassigned immediately to the number of tasks in processor i that will use the new copy of m. If task x does not use m but produces m, there should be no copy of data item m in $LM(i)$ before executing task x.

Theorem 6.5 *In $LM(i)$, a data item copy m becomes dead iff $usage(m) = 0$.*

Proof: If data item copy m is dead at time p after the execution of task x, no tasks after that time in processor i will use that copy, thus $usage(m) = 0$.

If $usage(m) = 0$ at time p, we claim that no task will use m after p. Assume that copy m is received at time t and after that there are n tasks using that after time t. Then $usage(m) = n$ at time t. If some task x is using that copy m after p, according to Theorem 6.3, tasks between time t and p that use data item m must use the same copy. Thus there should be less than n tasks referring that copy. Then $usage(m) > 0$ after time p since each reference will decrease $usage(m)$ by 1. This is a contradiction. □

When a task y (or host) sends m to processor j, the initial counter value of $usage(m)$ can be determined by counting the number of successors of task y (or host) in processor j using that m. That $usage(m)$ value is sent to processor j with data item m.

Example: In Table 6.2, the left column is one execution trace (receiving of data items and execution of tasks) on processor 1 in Figure 6.1(b). The right column is the new *usage* values in $LM(1)$ after the operation in the left column is completed. When the *usage* value of a data item becomes 0, the copy of this item is deleted from the local memory.

6.4 Iterative Execution of a DAG

We have analyzed the correctness of executing a DAG in one iteration on message-passing multiprocessors. In this section we examine how to deal with the iterative execution of a DAG.

In many applications, a problem solving process can be expressed as executing the same DAG in an iterative manner. For example, Jacobi, Gauss-Seidel, and SOR

Trace in Proc 1	Changes in *usage*
receive 1	usage(1)=2
receive 4	usage(4)=1
execute T_1^4	usage(1)=1, usage(4)=1
receive 5	usage(5)=1
receive 2	usage(2)=2
execute T_2^4	usage(2)=1, usage(4)=1
execute T_1^5	usage(1)=0, usage(5)=1
execute T_2^5	usage(2)=0, usage(5)=1
receive 3	usage(3)=1
execute T_3^5	usage(3)=0, usage(5)=0

Table 6.2: An example of run-time detection of dead data items.

methods for solving linear systems. The number of iterations is either unknown at compile-time or too large to produce the entire static DAG. We can derive a schedule of a DAG for one iteration and then repeatedly execute this schedule.

Given a task list $TaskExec(i)$ for processor i, it executes tasks one by one from the first to the last in this list for the current iteration and then it starts to execute the first task again as soon as it finishes the last task for the current iteration. In this way, execution of different iterations can be overlapped and their schedules be pipelined.

Figure 6.9(b) is a schedule for a DAG in (a). Figure 6.9(c) illustrates the iterative execution of this DAG and (d) is the pipelined schedule for (c). Dashed arrows between iterations in Figure 6.9(c) depict the dependence between iterations. Since data for different iterations flow around network and there could be communication between iterations, special care needs to be taken for code generation to make sure that each task at each iteration receives the correct data items.

For iterative execution of a DAG G, the user's input is a set of tasks in G, the data items communicated between tasks within the same iteration and data items communicated between the same or different tasks of different iterations. We have shown that if a DAG satisfies condition DA, the previous method for asynchronous execution of a DAG is correct. We further impose the following condition to ensure the correctness of the iterative execution.

IDA: *(1) A task receives distinct data items from its predecessors of the same or*

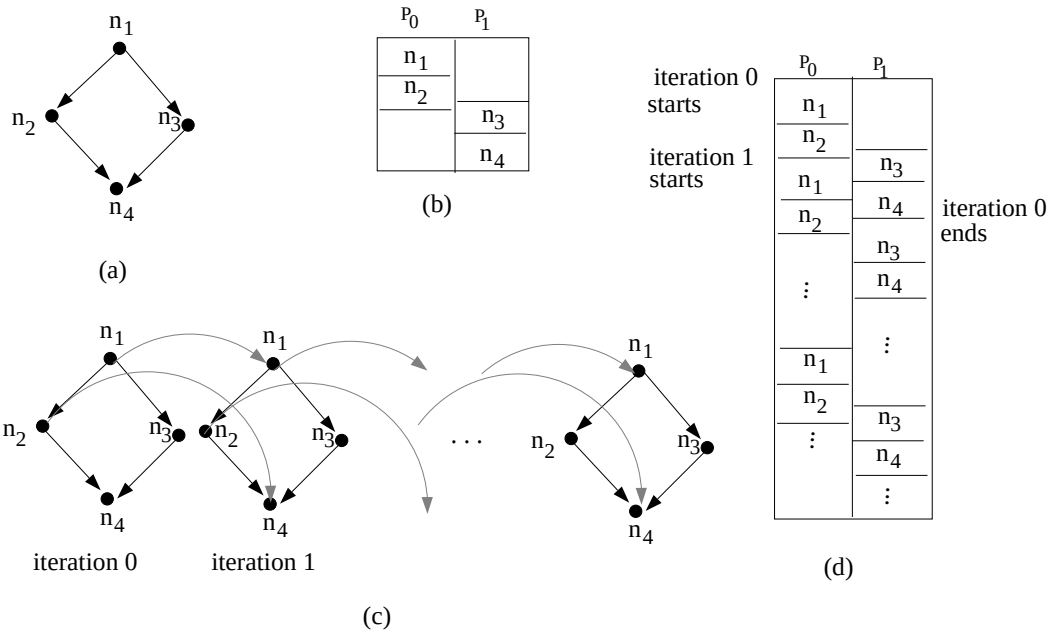


Figure 6.9: Iterative execution of a DAG.

previous iterations. (2) If a task x produces data item m for the use of tasks at later iterations, then there is no task other than x that also writes m. (3) Also task x should not produce data m at different iterations for the uses of tasks at the same iteration.

In the implementation for iterative execution, we incorporate the iteration number in data naming so that data used in different iterations can be distinguished. Data item m produced by task x at iteration k is now renamed as (m, j) to be used at iteration j $(j \geq k)$

Theorem 6.6 *If DA for G and IDA are true, then the iterative execution of G is correct.*

Proof: Let k be the iteration number. Let Max_Iter be the total number of iterations. We can consider the iterative execution of a DAG G as executing an expanded DAG IG which is obtained by duplicating G Max_Iter times as shown in Figure 6.9(c). In this proof, we distinguish tasks at different iterations by associating the iteration number k with their original names. Task x executed at iteration k is renamed as (x, k). The expanded graph IG is a DAG.

We first examine if the expanded graph IG satisfies DA and then check if the

schedule for IG is valid. Thus we can directly apply the results of Theorem 6.3 and 6.4 to IG, which will prove our statement.

By assumption DA1 for G and IDA(1), IG satisfies DA1.

Assume that a task (x,k) at iteration k produces data item (m,j) $(j \geq k)$ and task (y,j) uses a data item also called (m,j). Suppose that data (m,j) used by (y,j) is not produced by task (x,k). If $k = j$, then there is a dependence path between (x,k) and (y,k) by DA2 for G. If $k \neq j$, by IDA, no task with name other than x can produce data m. Assume (x,k') $(k \neq k')$ produces data item (m,j) for task (y,j), which is impossible by IDA(3). Thus IG satisfies DA2.

Assume that a task (x,k) writes data item (m,j) $(j \geq k)$, another task (y,i) at iteration i also produces data item (m,j) $(j \geq i)$. If $k = j$ and $i = j$, then there is a dependence path between (x,k) and (y,k) since this is the case by DA3 for DAG G. If $k \neq j$, then by IDA(2) and (3), this case is impossible. Thus IG satisfies DA3.

We examine DA4 for IG based on Theorem 6.2. Suppose task (x,k) and (z,i) both produce data (m,j) where $i \leq j$ and $k \leq j$, and task (y,j) uses data (m,j) produced by (x,k). We need to show it is impossible that there is a dependence path going from (x,k) to (z,i) and then to (y,j). Suppose that is the case. Then if $x = z$ and $k \neq i$, it is impossible by IDA(3). Now assume $x \neq z$. If $k < j$ or $i < j$, it is impossible by IDA(2). Thus if $k = j$ and $i = j$, it is impossible by DA4 for G. Since all cases are impossible, we conclude that IG satisfies DA4.

Finally the iterative schedule for IG is legal since each processor will not execute tasks for next iteration until the completion of task execution for the current iteration.
□

6.5 Multicasting on Hypercubes

We have used a multicasting function $multicast(m, D)$ for the code generation. In this section we present an algorithm for implementing multicasting on a hypercube. For one-to-all broadcasting such that every processor in a machine needs this message, a minimum spanning tree algorithm on a hypercube has been developed by Saad and

Schultz [88]. However as we will see from the discussion in the next subsection this algorithm is not suitable for multicasting where not every processor needs to receive a message. We will present an efficient algorithm for multicasting. The key idea in our multicasting algorithm is to eliminate unnecessary forwarding processors by reducing multicasting range from the entire hypercube into several smaller subgraphs which sufficiently cover all destinations. Section 6.5.2 describes the structure of subgraphs in a hypercube. Section 6.5.3 gives a method that partitions an entire hypercube into several small subgraphs for multicasting.

6.5.1 Preliminaries

We consider a hypercube with dimension n. The total number of processors is $p = 2^n$. A processor node can be represented by a binary number. Two nodes are connected if their binary numbers differ in one bit.

Let $H(p_1, p_2)$ be the Hamming distance between two nodes, i.e. the number of bits differing from each other in their binary numbers. There exists a path from p_1 to p_2 with a distance equal to $H(p_1, p_2)$. Processor nodes in that path can be found by toggling the bits of p_1 from left to right until reaching p_2. We call such a routing path *left-toggling Hamming path*, LH path for short.

The processor that issues $multicast(m, D)$ is called the *source*. Assume that the source is s and destination processors are $D = \{p_1, p_2, \cdots p_k\}$. The requirement for a multicasting algorithm is that *every node p_i receives the message within delay distance $H(s, p_i)$. Also every processor node in a multicasting scheme should be visited at most once for message routing.* A multicasting algorithm satisfying such constraint is also called a *shortest path multicasting* algorithm.

Since communication between processor nodes is *asynchronous*, a packet-switching communication scheme is needed. Namely, a message packet is sent from a source to a node and this node forwards this packet to another node. Finally the packet reaches the destination. We have assumed that communication between nodes is reliable and error-free.

Saad and Schultz [88] have developed a spanning tree algorithm for broadcasting

in a hypercube. A node in a tree forwards a message to its child nodes by toggling successive bits in a direction from the least-significant to most significant. We call this tree as *left-toggling spanning tree*, L tree for short. Figure 6.10 is an example of L spanning tree broadcasting from node 0000.

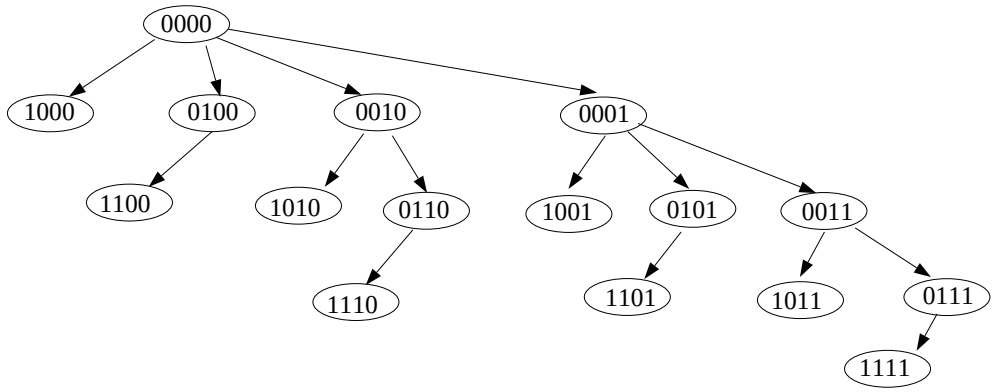


Figure 6.10: A L spanning tree of a hypercube of dimension 4, least significant to most significant.

A trivial extension of Saad and Sultz's algorithm for multicasting is as follows.

Algorithm A: we associate destination information in the control header of a broadcasting message packet such that a node receives a message and copies the message if it belongs to that destination set. Afterwards it forwards this message to its successors in a L tree.

Apparently algorithm A is a shortest path multicasting algorithm. We now discuss the performance issues in designing multicasting algorithms. For one-to-some multicasting, a routing mechanism to achieve selective communication may require some non-destination processors to be involved message forwarding. We call this *forwarding overhead*. For example, in algorithm A no matter how small the destination set is, all nodes are involved message forwarding. We need to develop a shortest path multicasting scheme with small forwarding overhead. We measure the effectiveness of selective multicasting using the following ratio:

$$R = \frac{|D|}{\text{Total number of nodes visited excluding } s}.$$

Example: We use algorithm A to broadcast from $s = 0000$ to $D = \{0001, 1100, 1101,$

$1110, 1111\}$. As shown in Figure 6.10, every destination x in D will receive the broadcasted message within the delay of $H(0000, x)$ and every node is visited once. The effective ratio is $R = 5/15 = 0.33$.

As we will see that improving the effectiveness of selective broadcasting could substantially reduce network traffic and alleviate communication congestion. We need to introduce the following basic operators for designing our multicasting algorithm. Each of the following operators is counted as one time unit in the complexity analysis.

- Arithmetic operators: $+, -, *$.
- Bit operators: $\sqcap, \sqcup, \neg, \oplus$ corresponding to bit AND, OR, NOT and XOR.

Let the binary identification number of a processor x be $b_{n-1}b_{n-2}\cdots b_0$ where b_j is the j-th digit. The following two functions will be used:

- $rightmost1(x) = 2^j$

  where the j-th digit is the least significant digit in x that has bit 1. This can be computed by $x \oplus (x \sqcap (x-1)))$ and thus its time complexity is $O(1)$.

- $leftmost1(x) = 2^j$

  where the j-th digit is the most significant digit in x that has bit 1. The time complexity for this function is $O(\log p)$.

In the current hypercube machine such as nCUBE-2, the wormhole transmission of a message can be used so that the distance of two processors ($H(p_1, p_2)$) does not play a significant role in the communication latency if network communication load is not heavy [32]. For wormhole communication between p_1 and p_2, a wormhole path is created between them and this path uses the channels between the intermediate routing processors. Since channel resource between two negbhor processors is limited (usually 1), even the message transmission between p_1 and p_2 becomes faster, the occupation of channels by this wormhole transmission could block other communication in the network. In nCUBE-2, the wormhole routing path between p_1 and p_2 are the LH path between them.

We will use the wormhole sending of a message from p_1 to p_2, if no intermediate processors in the LH path from p_1 are in set D. Our algorithm is based on left-toggling path and tree, to be consistent with the routing scheme of wormhole communication in nCUBE-2 so that no contention is created due to the use of wormhole message transmission. Other commercial machines use the similar routing scheme and our algorithm can be adapted easily.

6.5.2 Subgraphs for multicasting

We describe two types of basic subgraphs in a hypercube that can cover destinations in multicasting.

Linear path

If a destination set is simple enough, all destinations are in one path. There are many linear paths in a hypercube, we use a LH path. For example, in Figure 6.10, the LH path from 0000 to 1101 is $(0000, 0001, 0101, 1101)$. The advantage of this communication is its simplicity. The disadvantage is that there are few instances that D can be covered by a LH path.

Figure 6.11 is a procedure with a complexity of $O(|D| \log p)$ that determines if a LH path can cover all destinations in D. Such a path is called $PATH(s, D, x, pm)$ where x is the first destination node in this path and pm is a *path mask*. A bit in the binary number of pm is 1 if this bit is allowed to be toggled in routing.

1. Let x be the nearest node in D from s. Let y be the farthest node in D from s.
2. Let *path mask* be $pm = s \oplus y$.
3. For each node z in D, check if $z \in PATH(s, D, x, pm)$ using the following condition

$$pm \sqcap (\neg(2 * leftmost1(s \oplus z) - 1)) \sqcup (s \oplus z) = pm.$$

Figure 6.11: The procedure of finding a path to cover destinations.

Example: In Figure 6.10, assume $D = \{0011, 0111, 1111\}$. Then the routing path is $PATH(s, D, x, pm) = (0000, 0011, 0111, 1111)$. The effective ratio is $R = 1$.

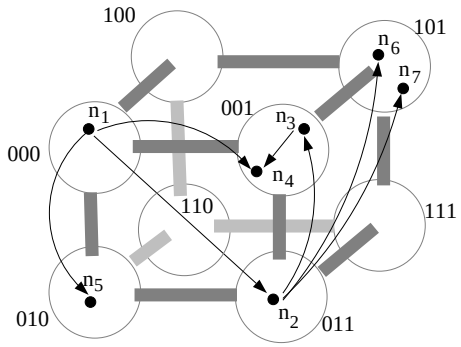


Figure 6.12: Part of a DAG mapped to a hypercube with 8 nodes.

Example: Figure 6.12 shows a part of a DAG mapped onto a hypercube of dimension 3. The multicasting from n_2 to n_3, n_6 and n_7 can be implemented as a linear-path communication along processor path 011, 001 and 101.

Subcube

A hypercube can be decomposed into subcubes with smaller dimensions. We may limit the multicasting range in a subcube, which could substantially reduce forwarding overhead.

We represent a subcube with a known member y using a bit mask m, called *subcube mask*. The i-bit of m is 0 if all nodes in that subcube have the same value in that bit, otherwise 1. The number of 1 bits in m is the dimension of that subcube. For example, a subcube with a node $x = 0000$ and mask $m = 0110$ is 0000,0100,0010,0110. It is easy to verify that a processor node z belongs to that cube if $z \sqcap \neg m = x \sqcap \neg m$.

Example: In Figure 6.12 task n_1 broadcasts a data message to n_2, n_4 and n_5. This communication can be accomplished using subcube broadcasting within the cube composed of processor 000, 001, 011 and 010 where the subcube mask is $h = 011$.

Given a set of nodes $S = \{n_1, n_2, \cdots, n_t\}$, the subcube with mask $m(S)$ that contains all those nodes is called the *minimal* subcube of S and it is determined as $m(S) = (n_1 \oplus n_2) \sqcup (n_1 \oplus n_3) \sqcup \cdots \sqcup (n_1 \oplus n_t)$. The time complexity is $O(|S|)$.

Given source s and a destination set D, we can derive a subcube mask $m(D \bigcup \{s\})$ that covers the source and all destination nodes.

Example: In Figure 6.10, $s = 0000$, $D = \{0100, 0010, 0110\}$. Algorithm A has

effective ratio $R = 3/15 = 0.33$. Using subcube mask $m(D \bigcup \{s\}) = 0110$, multicasting is limited in a subcube of dimension 2. The new effective ratio is $R = 3/3 = 1$.

It is easy to see that *the minimal subcube of D is always contained in the minimal subcube of D' if $D \subset D'$.* Thus we may further reduce multicasting range by excluding s from subcube computation: first we determine the minimal subcube mask $m(D)$ that only covers destinations. Then in multicasting, s first sends a message to a *delegate* source ds in that subcube using wormhole direct sending. And the message is further forwarded from ds to others within that subcube. The delegate source is determined as: $ds = (s \sqcap m(D)) \sqcup (x \sqcap (\neg m(D)))$ where x is any member of D.

We call the minimal cube of D as $CUBE(s, D, ds, m)$ with delegate ds and mask m. Every node y in this cube will receive a message within delay $H(s, y)$ even this message is routed through delegate ds. This is shown by the following result.

Theorem 6.7 *1) $ds \in CUBE(s, D, ds, m)$.*
2) If $s \in CUBE(s, D, ds, m)$, then $s = ds$.
3) If $s \notin CUBE(s, D, ds, m)$, then $s \neq ds$ and for any node $y \in D$, $H(s, ds)+H(ds, y) = H(s, y)$.

Proof: Let m stand for $m(D)$, and $CUBE$ stand for $CUBE(s, D, ds, m)$.

1) Since $ds = s \sqcap m \sqcup x \sqcap \neg m$ and $x \in CUBE$,

$$ds \sqcap \neg m = (s \sqcap m \sqcap \neg m) \sqcup (x \sqcap \neg m) = x \sqcap \neg m.$$

Thus $ds \in CUBE$.

2) If $s \in CUBE$, then $ds \sqcap \neg m = s \sqcap \neg m$. Also,

$$ds \sqcap m = (s \sqcap m) \sqcup (x \sqcap \neg m \sqcap m) = s \sqcap m.$$

Thus $ds = s$.

3) If $s \notin CUBE$, but $ds \in CUBE$, then $s \neq ds$.

For any node $y \in D$, $ds \sqcap \neg m = y \sqcap \neg m$. Also $ds \sqcap m = s \sqcap m$. Thus

$$H(s, ds) = H(s \sqcap \neg m, ds \sqcap \neg m) = H(s \sqcap \neg m, y \sqcap \neg m)$$

and

$$H(ds, y) = H(ds \sqcap m, y \sqcap m) = H(s \sqcap m, y \sqcap m).$$

Thus

$$H(s, ds) + H(ds, y) = H(s \sqcap \neg m, y \sqcap \neg m) + H(s \sqcap m, y \sqcap m) = H(s, y).$$

□

Example: In Figure 6.10, assume $s = 0000$ and $D = \{1001, 1101, 0011, 0111, 1111\}$. $m(D \bigcup \{s\}) = 1111$ with effective ratio $R = 5/15 = 0.33$. We compute the subcube mask of D: $m(D) = 1110$ and $ds = 0001$. Then 0000 first sends a message to 0001 and then this message is further broadcasted from 0001 within that subcube. The new effective ratio is $R = 5/8 = 0.63$.

6.5.3 The elimination algorithm for improving the effective ratio

We have described two subgraph structures to cover a destination set. However it is impossible that a destination can always be covered using one path or one subcube with high effective ratio. For example, if $s = 0000$ and $D = \{0001, 0010, 0100, 1000\}$, no path can cover all destinations. The minimal subcube of D is still the entire cube with dimension 4. Thus we need to partition the given destination set D into a set of subsets such that each subset can be covered by a graph structure. In this way, useless processors will be eliminated from multicasting and a high effective ratio can be achieved.

Since D can be covered by a L spanning tree, we use subtrees of the root in this tree to partition destinations. We describe the topology of a L spanning tree and the subtrees of its root as follows: the L tree of a hypercube of n dimension is a *binomial* tree of degree n. A binomial tree of degree n is called T_n for short. The recursive definition of T_n is: 1 node is T_0. T_n is obtained by adding an edge between the roots of two T_{n-1} trees.

Deleting edges from the root of T_n to its immediate child nodes results in n *disjoint* binomial subtrees: $T_0, T_1, \cdots, T_{n-1}$. The root of each subtree differs in one bit from

Let *mynode* be the current processor number.
Let m be the received subcube mask and D_m be the received destination set.
While $m > 0$ **Do**
  $right = rightmost1(m)$; $sr = mynode \oplus right$.
  Find the set of destinations D_{sr} in D_m that belong to the subtree of root sr.
  If $D_{sr} \neq \emptyset$ **Then**
    If D_{sr} only has one node, **then** send the message directly to it.
    Else Use a LH path for D_{sr} if possible otherwise use the minimal subcube.
  EndIf
  $m = m \oplus right$.
Endwhile

Figure 6.13: The partitioning and routing procedure on a hypercube processor, which eliminates unnecessary forwarding processors for multicasting.

the root $r(T_n)$ of T_n. The rule for splitting destination set D into n subsets S_i is: *a destination x belongs S_i corresponding to subtree T_i if* $rightmost1(x \oplus r(T_n)) = 2^{n-i}$.

We now describe the elimination algorithm for implementing procedure $multicast(s, D)$ on a hypercube.

Algorithm E: Processor s first tries to use a L path to perform multicasting. If impossible, then it computes the minimal subcube of D and use the subtree partitioning procedure in Figure 6.13 to eliminate unnecessary processors.

In Figure 6.13, each iteration of **While** loop finds a subset D_{sr} corresponding to a binomial subtree. If the subset is empty, the corresponding subtree is eliminated. When a LH path is found to cover D_{sr}, processor *mynode* sends the message to the nearest node in that path. This message is further forwarded to the next nearest node and so on. Thus non-destination nodes in that path are skipped. When the minimal subcube $CUBE(mynode, D_{sr}, ds, m)$ is used for D_{sr}, processor *mynode* forwards the message with cube mask m to delegate ds. At node ds, the procedure in Figure 6.13, is called to further disseminate a message within that subcube.

The time complexity of this elimination algorithm at each routing processor is $O(|D| \log p)$ and the space overhead is $O(|D|)$.

Example: In Figure 6.10, assume $s = 0000$, $D = \{1100, 0110, 1110, 0011, 1011, 0111\}$. The elimination algorithm results in $R = 1.0$ for this case as explained below.

At node 0000, D is split with respect to binomial subtrees (T_3, T_2, T_1, T_0). T_3 has

3 nodes $D_3 = \{0011, 1011, 0111\}$. Its minimal subcube is $CUBE(0000, D_3, 0011, 1100)$. T_2 has 2 nodes $D_2 = \{0110, 1110\}$, it can be covered by a path. T_1 only has one node 1100 and a message is directly forwarded to 1100. T_0 does not contain any destination and is eliminated.

When a message packet with destination information D_3 arrives at node 0011, this node further splits destinations into $\{1011\}$ and $\{0111\}$ and each of them can be reached by directly sending a message.

Theorem 6.8 *In the elimination algorithm for multicast(s, D), every routing node (including those implicitly used by wormhole message transmission) is visited once and each destination x receives the message within distance delay $H(s, x)$.*

Proof: We show it by induction on the dimension of minimal subcube of D, called $d(m(D))$ for short.

When $d(m(D))$ is 0 or 1, it is trivial and a L path will cover D. Assume the statement is true for $d(m(D)) < k$.

When $d(m(D)) = k$, if the elimination algorithm finds a path to cover, then the statement is true. If not, the partitioning procedure in Figure 6.13 is applied to the minimal subcube of D. For each subset D_{sr} corresponding to a binomial subtree, $d(m(D_{sr})) < k$. By the induction hypothesis, each destination x in D_{sr} will receive the message within delay $H(ds, x)$ from ds where ds is the delegate of that subcube. By Theorem 6.7, $H(s, ds) + H(ds, x) = H(s, x)$, sending the message to this delegate and then forwarding within the subcube does not increase the delay distance. Thus every destination x in D_{sr} will receive the message within delay $H(s, x)$ from s.

Since binomial subtrees of s are disjoint, it is impossible that the message is sent more than once to the same delegate in a subtree. Also the wormhole routing is consistent with L tree. No intermediate processor used in a wormhole LH path will be used more than twice by other wormhole communication. Thus by the induction hypothesis, each routing processor will be visited at most once. □

6.5.4 Experiments on multicasting

We examine the performance of the elimination algorithm and algorithm A. We call the elimination algorithm as E in this subsection. We first describe an experiment on nCUBE-II that verifies the importance of multicasting with high effective ratio. Then we report the effective ratio of E and A on randomly generated cases.

Experiment 1: We use $p = 2^n$ nCUBE-II processors and processor node i is multicasting a M-byte message i to all nodes in a subcube with source i and subcube mask $2^{n-1} - 1$. Namely, only half of processors in the entire hypercube need to receive message i sent from processor i. We select message size $M = 0.1K, 1K, 4K, 10K$ and $p = 4, 8, 16, 32, 64$.

We use algorithm A and E for multicasting. In algorithm A, a processor x is a recipient of message i if $(x \oplus i) \sqcap \neg(2^{n-1} - 1)) = 0$. The effective ratio of algorithm A is $R = 0.5$. Algorithm E can correctly identify the subcube that needs a message. The effective ratio of E is $R = 1$.

We measure the completion time T_A and T_E of the last processor. The improvement ratio of E over A is computed as $1 - T_A/T_E$. Table 6.3 lists improvement ratios for different values of M and p on nCUBE-II.

M=	0.1K	1K	4K	10K
p=4	10.0	10.5	11.8	12.6
p=8	17.8	14.5	11.8	12.9
p=16	15.1	10.8	13.7	14.2
p=32	15.5	10.2	12.0	12.8
p=64	15.2	9.4	11.4	11.9
Average	14.7	11.8	12.1	12.7

Table 6.3: Improvement of E over A on nCUBE-II when $R(A) = 0.5$.

The completion time is found to be proportional to M, p and effective ratio R. Given M and p, the average improvement ratio of method E over A is about 13% for this case. When we change the subcube mask as $2^{n-2} - 1$ using $p = 8, 16, 32, 64$, the effective ratio of method A is 0.25. The average improvement ratio of E over A is about 26% as listed in Table 6.4.

M=	0.1K	1K	4K	10K
Average	29.5	24.1	26.1	26.7

Table 6.4: The average improvement on nCUBE-II when $R(A) = 0.25$.

Experiment 2: We examine the effective ratios of algorithm E and A on randomly generated cases. We randomly generate 100 multicasting cases for a given p and the number of destinations varies between $3 < |D| < p$. The result is listed in Table 6.5.

	A	E
p=8	0.69	0.97
p=16	0.52	0.93
p=32	0.48	0.91
p=64	0.48	0.90
p=128	0.48	0.88
p=256	0.55	0.89
p=512	0.47	0.86
p=1024	0.52	0.87

Table 6.5: The average effective ratio of E and A on random cases. $3 < |D| < p$.

To examine effective ratios for smaller sizes of destination sets, we also randomly generate 100 cases with $3 < |D| < p/2$. The result is listed in Table 6.6.

	A	E
p=16	0.31	0.97
p=32	0.27	0.90
p=64	0.26	0.85
p=128	0.26	0.83
p=256	0.27	0.83
p=512	0.25	0.82
p=1024	0.25	0.80

Table 6.6: The average effective ratio of E and A on random cases. $3 < |D| < p/2$.

From this experiment we conclude that the average effective ratio of the elimination algorithm is high and has a substantial improvement over algorithm A.

Lan, Esfahanian and Ni [65] have also proposed a multicasting algorithm for hypercube, our algorithm uses subgraph elimination that could reduce more unnecessary

forwarding processors comparing with their algorithm. McKinley et. al [70] have proposed a new contention-free algorithm, called the *U-cube tree* algorithm that takes the advantage of wormhole routing so that no forwarding overhead is involved. To achieve that, this algorithm assumes that it takes the same amount of time to transmit a message between any pair of processors in the network, and also it requires that two messages can be transmitted simultaneously in opposite directions. Our algorithm is also contention-free but it is not assumed that wormhole transmission takes the same time as neighbor communication, and any pair of nodes involves one direction transmission of a message once during the entire multicasting process. In this way, considerably less amount of wormhole-routed communication is used in our algorithm, which could reduce the chance of blocking other communication conducted in the network and hence reduce the communication contention. However, there exists forwarding overhead in our multicasting algorithm and more investigation of its performance is needed.

6.6 Related Work and Remarks

We have described the generation of SPMD code that incorporates performance improvement techniques without causing deadlock and data inconsistency. This code ensures a correct program execution in a distributed memory machine for arbitrary DAG parallelism with a shared memory programming style. Several other groups have also developed compiler code generation techniques for message-passing architectures. We discuss some related work.

Koelbel and Mehrotra [61] have proposed a code generation method for compiling DOALL loop parallelism (no dependencies across loops) which allows program segments to access data items using a global name space. Our work is more general in the sense that we address general DAG parallelism and also allow global data naming.

Li and Chen [67] in their CRYSTAL project have proposed the code optimization using aggregate communications. We have incorporated the idea of aggregate communication idea used in [67]. One difference is that we use asynchronous instead of synchronous communication. We have also developed techniques for eliminating redundant communication and maintaining data coherence, and have incorporated them in

one execution scheme.

Hiranandani, Kennedy and Tseng [51] and Rogers and Pingali [86] have described several communication optimizations for Fortran D and Id Nouveau. Message vectorization combines several small messages into a big message to reduce startup overhead. However, it is hard to judge when vectorization should be used or not since message pipelining could also improve performance. Currently we have not incorporated the automatic vectorization. We expect that a DAG is derived based on coarse grain parallelism and messages between tasks have large sizes.

Our work bears resemblance to that for shared memory machines. Shasha and Snir [92] described the conditions for ensuring correct execution of parallel programs that share memory. We have developed a scheme for correct execution of parallel tasks for distributed memory architectures. We have also presented the properties of a DAG as a condition to ensure this correctness.

Data coherence is related to cache coherence. Cheong and Veidenbaum [18] described data flow analysis techniques to detect stale data copies (a data copy in a processor cache does not have the up-to-date value) for shared memory architectures. In our case, a dead copy could be stale but also could be because no tasks access this copy. Further more, we use run-time resolution for enforcing data coherence.

Chapter 7

Performance Results

In this chapter, we present experiments with PYRROS on the nCUBE-II hypercube. In section 7.1 and 7.2 we consider the performance of PYRROS using two regular DAGs, the Gauss Jordan algorithm and the LU decomposition. PYRROS performs very well for both examples. In Section 7.3 we examine the sensitivity of PYRROS when inaccurate weight information is used as input. For coarse grain graphs there is small variation of performance as long as the input weights result in a coarse grain graph. For fine grain DAGs the performance variation could be significant.

In Section 7.5 we consider the classical Fast Fourier Transform (FFT) algorithm to demonstrate the importance of good partitions that have sufficient parallelism. The partition for FFT is based on a a recursive SPLIT and MERGE tree and the performance is bounded by the length of the critical path. PYRROS comes close to this lower bound but its performance is limited. In Section 7.4 we consider the matrix matrix multiplication to investigate the impact of data initialization and loading on overall performance.

One of the advantages of PYRROS is that it can handle irregular task graphs, which are very common in many applications. For example the solution of sparse matrix equations results in irregular task graphs. In Section 7.6 we present some preliminary experiments with PYRROS in solving a sparse matrix system. PYRROS performs very well for such task graphs. Another important application of PYRROS is iterative methods. Given an initial task graph, then an iterative method repeatedly executes a schedule of the initial task graph. In Section 7.7 we consider the solution of the Laplace equation based on the five point finite difference approximation and the Gauss-Seidel iterative method. PYRROS achieves very good speedups for this example as long as

the task graph is coarse grain.

7.1 The Gauss Jordan Algorithm

GJ without pivoting. We first consider the GJ algorithm without pivoting based on sub-matrix BLAS-3 decomposition shown in Section 3.3 along with its dependence graph.

To compare the performance of PYRROS with other commonly used schedules for this task graph, we have written a program based on "owners compute rule" shown in Figure 7.1, Ortega [77]. The resulting clustering using this program is the natural linear clustering. The clusters are mapped to the processors using wrap mapping along the gray code ring of the hypercube, Moler [73] and Saad [87]. T_k^j uses block column k to modify block column j. T_k^j is in cluster j and is mapped to processor $p_j = j \bmod p$.

```
for k = 1 to N
    Receive column k.
   for j = k + 1 to N + 1
      if this processor owns column block j, then
         Execute T_k^j
         if j = k + 1, broadcast column block k + 1. endif
      endif
   endfor
end
```

Figure 7.1: A parallel node program for block GJ using the "owner computes rule".

In Table 7.1, we show the performance of the automatically generated code by PYRROS and hand-written program. The size of the matrix is 1000×1000 and the block size $r = 10$. This implies that the number of column blocks is $N = 100$ and the size of the graph is about 5000 tasks and 10000 edges. PYRROS takes only few seconds to produce the parallel code.

The sequential time for $n = 1000$ is 1150 seconds and is estimated from the execution of the hand-written program when $n = 500$ using one processor of the nCUBE-II and the *ncc* compiler version 2.2 with -O option. The problem size $n = 1000$ is too big for the 4 MB per node NCUBE-II machine. PYRROS first simulates the execution

on the SUN front end workstation using the nCUBE-II parameters and gives a predicted performance. For this DAG, the natural linear clustering is optimal as shown in Section 3.3. The performance of the automatically-generated code is comparable to that of our hand-written version. The predicted performance by PYRROS is very close to the real performance for this coarse grain DAG. In general, PYRROS will give good performance predictions provided the task graph is coarse grain. For small size problems, the weights cannot be estimated accurately due to various machine overhead involved in execution and as result PYRROS predictions will not be accurate.

	PYRROS		Hand	speedup
	predict	real		
p=4	299.0	283.4	288.7	4.0
p=8	155.7	150.5	150.4	7.6
p=16	84.5	81.0	81.4	14.1
p=32	50.0	48.7	47.4	24.3
p=64	33.0	31.8	32.8	35.1

Table 7.1: Parallel time of executing the GJ DAG with $n = 1000$ on nCUBE-II. Speedup for PYRROS is obtained using the sequential time of the hand-written program.

GJ with pivoting. Introducing pivoting makes the GJ program slightly more complicated. The programs are given in the Appendix. Figure 7.2 shows the speedup of PYRROS code over the sequential code for $n = 500$ and $n = 1000$ on nCUBE-II with four megabyte of memory per node. The sequential times for $n = 500$ with single and double precision are 164.7 and 187.1 seconds respectively. For $n = 1000$, the sequential time with single precision is an estimation from the $n = 500$ size, i.e. $8 \times 164.7 = 1317.6$ seconds.

PYRROS achieves very good speedups, but the performance depends on the size of problem, the degree of parallelism and the task graph granularity. This is clearly demonstrated in Table 7.2 where the speedup of PYRROS for different values of N and r is presented for $p = 64$. The entry for $n = 2000$ with double precision is not listed because of memory overflow.

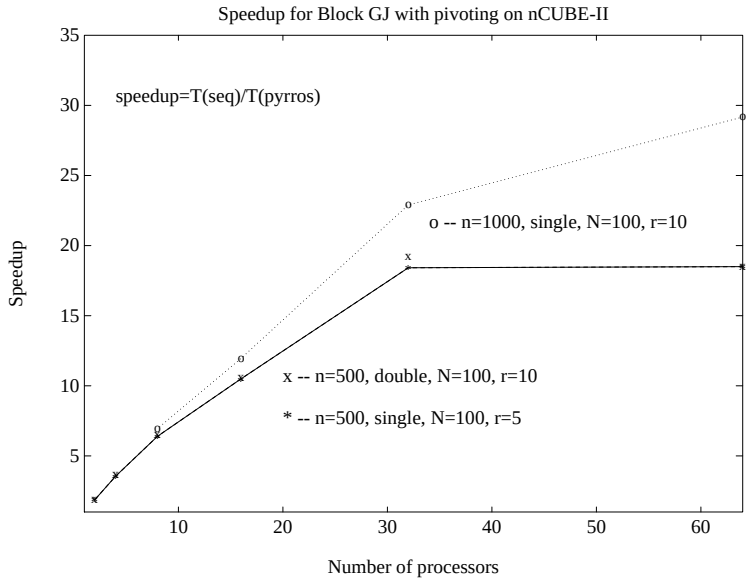


Figure 7.2: The speedup of PYRROS code for GJ with pivoting on the nCUBE-II.

	$N = 100, r = 5$ $n = 500$	$N = 100, r = 10$ $n = 1000$	$N = 200, r = 5$ $n = 1000$	$N = 125, r = 16$ $n = 2000$
single	18.5	29.1	41.1	37.1
double	18.5	27.6	38.4	–

Table 7.2: Speedup for block GJ with pivoting on a 64 processors of a 4 MB per node nCUBE-II.

7.2 LU Factorization

LU without pivoting. The BLAS-3 LU factorization without pivoting is shown in Figure 7.3. The $n \times n$ matrix is divided into $N \times N$ submatrices and each submatrix has size of $r \times r$ where $N = n/r$. The resulting task graph is similar to GJ algorithm but the weight distribution differs. The dependence graph is shown in Figure 7.4. Each task T_k^j is operating on a block column composed of N elements and each element is a $r \times r$ submatrix. This graph is coarse grain in the top part but becomes fine grain in the bottom. The weights along with the PYRROS program are defined in the Appendix.

```
for k = 1 to N
  T_k^k : { Factorize A_{k,k} as L_k * U_k
        for i = k + 1 to N
            A_{i,k} = A_{i,k} * U_k^{-1}
        end}
  for j = k + 1 to N
      T_k^j : { A_{k,j} = L_k^{-1} * A_{k,j}
        For i = k + 1 to N
            A_{i,j} = A_{i,j} - A_{i,k} * A_{k,j}
        end}
  end
end
```

Figure 7.3: Block LU factorization and its task partitioning.

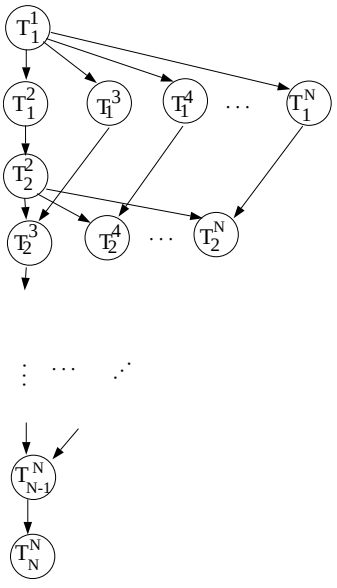


Figure 7.4: The task graph of block LU factorization with or without pivoting.

Figure 7.5 shows the speedup of PYRROS for $n = 450$ and $n = 1000$. For $n = 450$ the speedup for blocksize $r = 5$ becomes better than $r = 10$ as the number of processors increases. This is expected since the parallelism of the task graph reduces as the blocksize increases. Notice that there is very little difference when the number of processor is less than 8 since in this case there exists a sufficient number of parallel tasks.

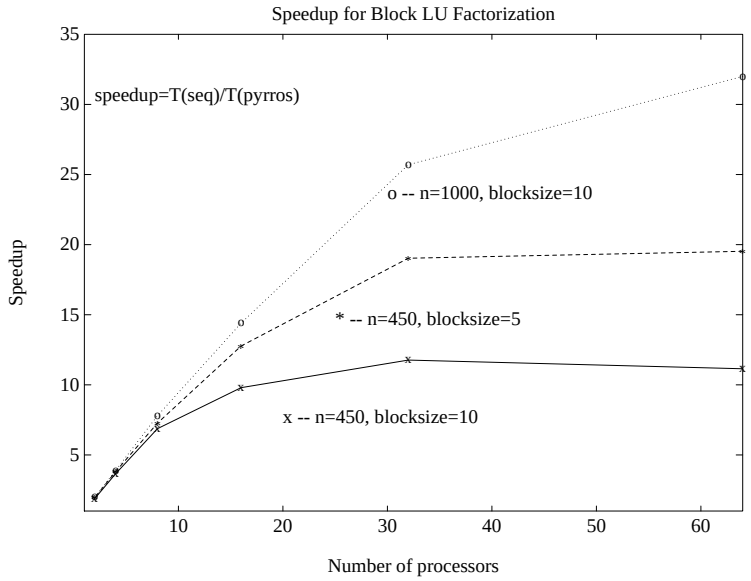


Figure 7.5: The speedup ratio of PYRROS over a sequential LU program.

LU with pivoting. The algorithm for LU with pivoting has the same dependence structure as LU without pivoting. The only difference is that task T_k^k needs to find an element for each column with the maximum absolute value. The row exchange information due to pivoting needs to be used in updating operation of T_k^j.

We have also written a C program based on "owners compute rule" shown in Figure 7.6. The resulting clustering using this program is the natural linear clustering. The clusters are mapped to the processors using wrap mapping along the gray code ring of the hypercube [77, 87]. T_k^j is in cluster j and is mapped to processor $p_j = j \bmod p$.

In Table 7.3, we show the performance of the automatically generated code by PYRROS and the hand-written program. The size of the matrix is 1024×1024 and the column block size $r = 8$. This implies that the number of column blocks is $N = 128$ and the size of the graph is about 8000 tasks and 16000 edges. PYRROS takes few

```
for k = 1 to N
   if this processor owns column block k, then
        Then do T_k^k and broadcast this data item.
   else receive the column block k.
   endif
   for j = k + 1 to N
      if this processor owns column block j,  then do T_k^j.
   endfor
end
```

Figure 7.6: A parallel node program for LU using the "owner computes rule".

seconds to produce the parallel code.

The sequential time for $n = 1024$ is 858.6 seconds with single precision and is estimated from the execution of the hand-written program when $n = 512$ using one processor of the nCUBE-II and the *ncc* compiler version 2.5 with -O option. The problem size $n = 1024$ is too big to execute in a nCUBE-II processor with 4 MB memory. PYRROS first simulates the execution on the SUN front end workstation using the nCUBE-II parameters and gives a predicted performance. The predicted performance by PYRROS is very close to the real performance for this coarse grain DAG. In general, PYRROS will give good performance predictions provided the task graph is coarse grain. For small size problems, the weights cannot be estimated accurately due to various machine overhead involved in execution and as result PYRROS predictions will not be accurate.

When $p \leq 32$, the performance of the automatically-generated code is comparable to that of our hand-written version but not for $p = 64$. This is because when p is small, each processor has the enough tasks to work. When $p = 64$, PYRROS scheduling algorithms play a role in optimizing the mapping and execution order to reduce the parallel time.

7.3 Sensitivity of PYRROS on Inputs with Inaccurate Weights

Parallel machines are complex systems and accurate estimation of communication and computation weight at compile time is very difficult. This is because of run time

	PYRROS		Hand	speedup
	predict	real		
p=4	225.6	225.5	225.3	3.8
p=8	120.5	120.8	120.7	7.1
p=16	68.2	69.1	68.6	12.5
p=32	42.5	44.2	42.7	19.4
p=64	24.1	24.0	29.7	35.7

Table 7.3: Parallel time of executing the LU DAG with pivoting for $n = 1024$ on nCUBE-II. Speedup is obtained by dividing the PYRROS real time over the sequential time.

factors, which are not possible to estimate them at compile time, such as network load, buffer size, cache hits and misses. Therefore, it is important to know the sensitivity of PYRROS on inaccurate weight inputs. In this section, we examine this issue using the GJ and LU DAGs. We first execute a DAG using its estimated weights and then vary the weights and measure the performance differences. We consider three categories of DAGs based on local task grain values: coarse grain, partially coarse grain, fine grain.

Coarse grain graphs: We start with the submatrix GJ DAG with $n = 1000$ and $r = 10$. This DAG has a constant grain value for each task. The computation weight of task T_k^j is $Nr^3\omega$, where ω is the time for performing $a = a + b * c$. For the nCUBE-II, we have estimated $\omega = 2.3\mu s$ for single arithmetic precision. The edge weights are $\alpha + Nr^2\beta$, with $\alpha = 200$ and $\beta = 2.4$. Thus the granularity is

$$g(T_k^j) \approx \frac{r\omega}{\beta} \approx 10.$$

We compare the parallel time using the normal weight assignment and the result when we use others. Figure 7.7 gives the difference ratio comparing the normal weights with three other cases. $wXcY$ stands for assigning computation weights as X and communication weights as Y. Thus for this coarse grain DAG, when we use coarse grain values, e. g. $wXc1$ and $X > 1$, the performance variation is very small, i.e. within 10%. Remember that for coarse grain task graphs DSC derives linear clusterings and that such a linear clustering is within 10% from the optimum clustering for this task graph, see Theorem 4.5. For $w1c1$, the performance difference is still within 35%. However when we treat it as a fine grain DAG using $w1c10$, the difference ratio varies from 30%

to 70%.

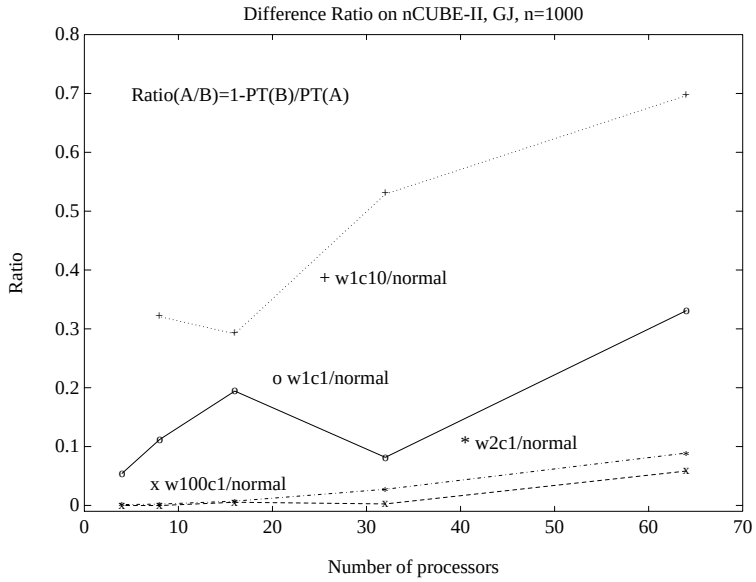


Figure 7.7: The impact of graph weight assignments on PYRROS performance for a coarse grain DAG. $wXcY$ stands for using X as task weight and Y as an edge weight.

Partially coarse grain graphs: Next we consider the block LU DAG when $n = 1000$ and $r = 10$. This DAG is partially coarse grain. For Figure 7.4, Task T_k^k has a weight $(N-k+1)r^3\omega/2$ and it communicates with T_k^j in $(N-k+1)$ submatrices. Task T_k^j has a weight $(N-k)r^3\omega$ and it sends column block j to T_{k+1}^j of size N submatrices. The top part of this DAG is coarse grain and the local grain values of tasks are greater than 1. The grain values become smaller from top to bottom. For single precision on the nCUBE-II,

$$g(T_1^j) = \frac{Nr^3\omega/2}{\alpha + Nr^2\beta} \approx \frac{r\omega}{2\beta} \approx 5$$

At the bottom part,

$$g(T_{N-1}^N) = \frac{r^3\omega/2}{\alpha + Nr^2\beta} \approx \frac{1}{20}.$$

Figure 7.8 gives the difference ratios compared the normal weight assignment with other five cases. For the small variation in task granularity such as using $w1c1$, the performance difference is within $\pm 20\%$. When we consider the original DAG very coarse by assigning $w10c1$ or $w100c1$, the difference is small if p is small since there is enough parallelism. The performance difference varies from -35% to 10% when p is large. However, when we regard this DAG as very fine by assigning $w1c10$ and $w1c100$, the

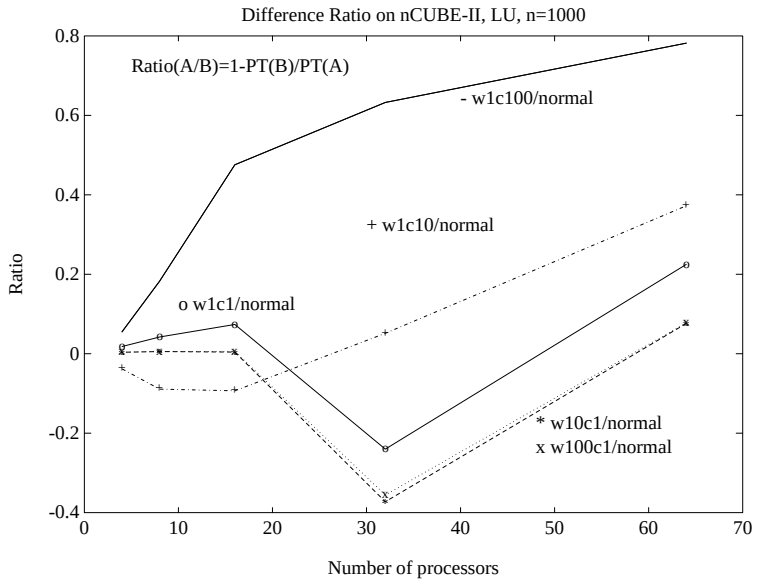


Figure 7.8: The impact of graph weight assignments on PYRROS performance for a partially coarse grain DAG.

performance difference could be up to 80%. In an extreme case, PYRROS would cluster all tasks in one processor if communication weights are too large. Notice again that the performance differences are small between coarse grain task weight assignments.

Fine grain graphs: Finally we examine the performance difference for a fine grain LU DAG when $n = 100$, $r = 1$ and $N = 100$. The local grain values are all less than 1. For task T_1^j and T_{N-1}^N we have that

$$g(T_1^j) \approx \frac{N\omega}{\alpha + N\beta} \approx \frac{1}{2}, \quad g(T_{N-1}^N) \approx \frac{\omega}{\alpha + N\beta} \approx \frac{1}{100}$$

Figure 7.9 gives the difference ratios compared the normal weight assignment with $w10c1$, $w1c1$, $w1c10$, $w1c100$,$w1c200$. The performance difference is from -5% to 30%.

7.4 Matrix Multiplication

In this experiment, we examine the impact of data initiation and loading in parallel processing. We consider the problem of matrix multiplication which multiplies two $n \times n$ matrices, $C = B * A$, as shown in Figure 7.10. Each T_i for $1 \leq i \leq n$ multiples matrix B with column i of A and produces row i of matrix C. Those tasks are independent and can be executed in parallel without inter-processor communication. Thus a linear

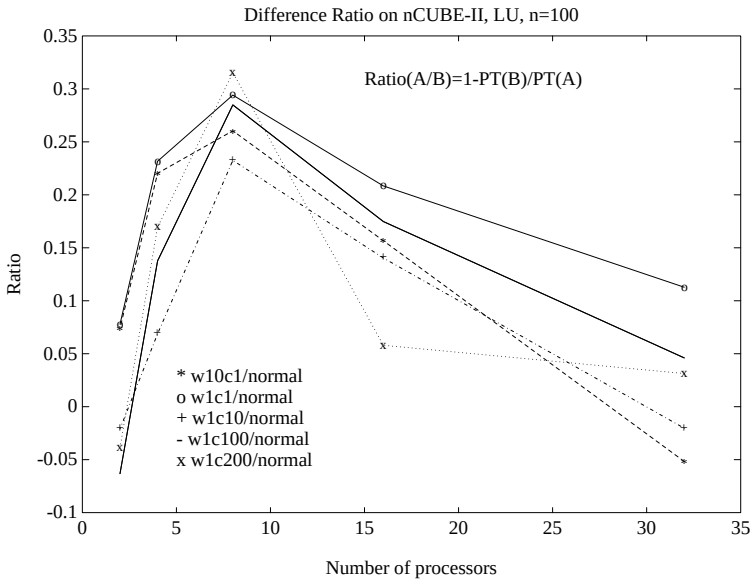


Figure 7.9: The impact of graph weight assignments on PYRROS performance for a fine grain DAG.

speedup can be obtained only if we do not include the cost of data loading or the initial data A and B are generated in each processor.

We assume that there is a task Y that initializes matrix B. Each task T_i owns matrix column i of A but needs to receive matrix B from task Y. The dependence graph is shown in Figure 7.11. We execute this DAG on the nCUBE-II and as we see in Figure 7.12 the linear speedup is not achieved. Observe that the speedup increases up to a point as the size of the matrix increases and then it becomes worst around 25 processors. This is an expected performance since for smaller number of processors computation dominates communication and data loading. For larger number of processors data loading becomes important because the size of the messages and distance that have to be transmitted increases.

7.5 Fast Fourier Transformation

We saw that PYRROS can achieve a good speedup in previous experiments. The performance, however, depends on the input partitioning and its granularity as well as the degree of the parallelism. The optimal speedup in executing a DAG is bounded

```
for i = 1 to n
    T_i : {
       for j = 1 to n
           c_{i,j} = 0
          for k = 1 to n
              c_{i,j} = c_{i,j} + b_{i,k} * a_{k,j}
          end
       end }
end
```

Figure 7.10: Matrix multiplication and its task partitioning.

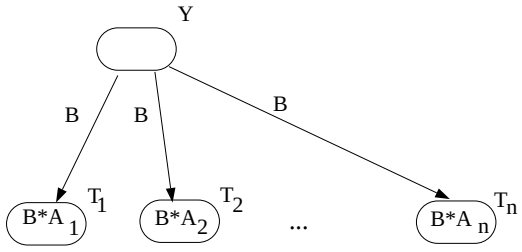


Figure 7.11: The task graph of matrix multiplication.

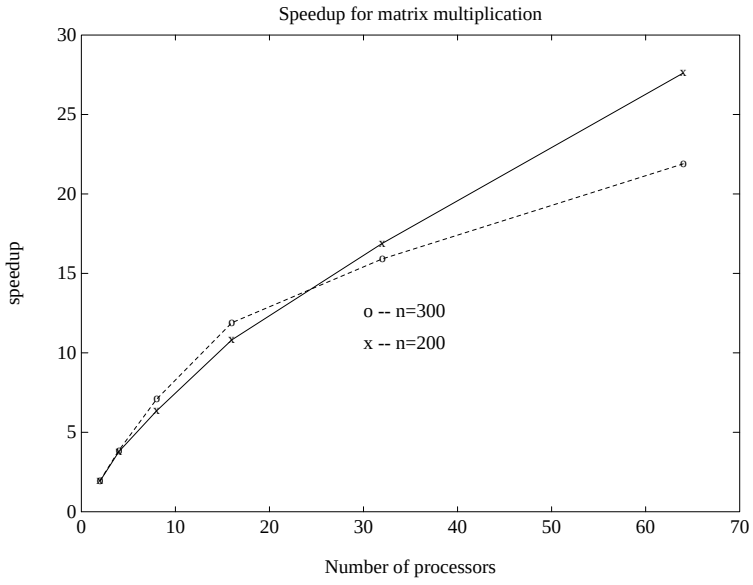


Figure 7.12: The speedup of matrix multiplication considering cost for data loading.

below by

$$\min(\text{Degree of parallelism}, \frac{T_1}{CPL})$$

where T_1 is the sequential execution time and CPL is the length of critical path without including communication. Notice that this bound is still loose since communication overhead has not been included.

The next example is the classical FFT transform of an array of complex numbers A with length $n = 2^m$ into a array B:

$$B_j = \sum_{k=1}^{n} A_k e^{-\frac{2\pi i}{n}(k-1)j}.$$

The FFT algorithm by Cooley and Tukey [25] can be considered as recursively computing the even and odd elements of A and then combining them together as shown in Figure 7.13. The dependence task graph can be considered as the concatenation of two trees: the MERGE tree and SPLIT tree. Each task in the SPLIT tree splits a vector into even and odd parts as shown in the line 2 of the FFT algorithm . Each task in the MERGE tree uses the result of two incoming data vectors of size $h/2$ Y and Z into a vector of size h as shown in the FFT algorithm, from line 4 to line 9. Figure 7.14 gives an example when $n = 8$.

$fft(A, B, n)$
1. **If** n is 1, **Then** { $B = A$; return. }
2. Let A_{odd} be the odd elements of A and A_{even} be the even elements of A.
3. **Call** $fft(A_{odd}, Y, n/2)$ and $fft(A_{even}, Z, n/2)$.
4. $c = cos(\frac{2\pi}{n}) + sin(\frac{2\pi}{n})i$; $w = 1$.
5. **for** $k = 1$ **to** $n/2$
6. $B_k = Y_k + w * Z_k$.
7. $B_{k+n/2} = Y_k - w * Z_k$.
8. $w = w * c$.
9. **Endfor**

Figure 7.13: Recursive FFT algorithm.

We execute this DAG on nCUBE-II when $n = 2^{14}$. Figure 7.15 shows the result of PYRROS code and the speedup does not scale up. The reason is that the task partitioning does not provide sufficient parallelism and also does not distribute arithmetic load evenly among tasks. The degree of parallelism for this case is $m = 14$. The root of

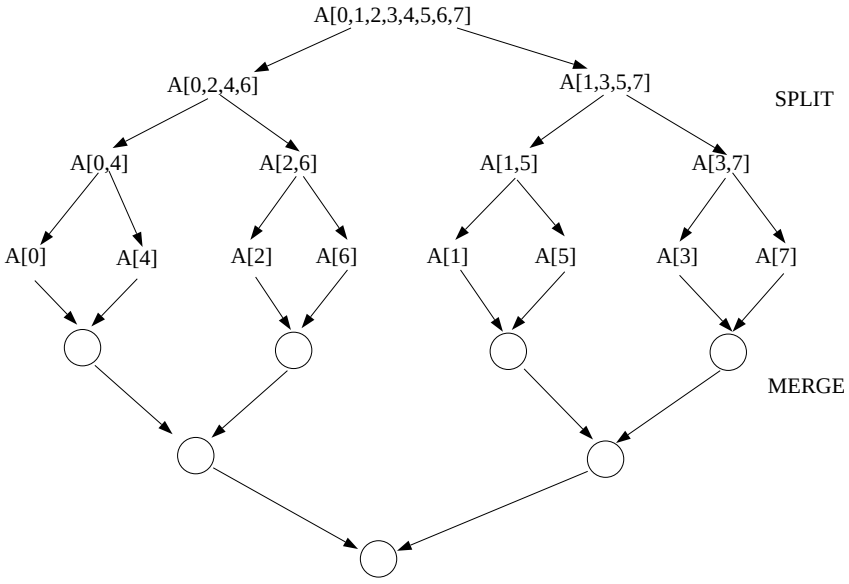


Figure 7.14: The task graph of FFT when $n = 8$.

the MERGE tree which combine large vectors has large arithmetic computation while while the leaf of tree has a small operation. More specifically, for $n = 2^m$, assume that a MERGE task, which combines two 2^h vectors into a 2^{h+1} vector, has a weight of $2^h(2 + 0.5)\omega$ where ω is the time for processing addition and multiplication of complex numbers. A SPLIT task, which divides a 2^h vector into two 2^{h-1} vectors, has a weight of $2^h\omega/4$. Then the length of critical path of this DAG without considering communication delay is:

$$CPL = (2^m + 2^{m-1} + \cdots + 2)\omega/4 + (1 + \cdots 2^{m-1})\omega 3/2 \approx 2^{m+1}\omega.$$

The total sequential time is:

$$T_1 = m2^m\omega/4 + m2^{m-1}3/2\omega = m2^m\omega.$$

The upper bound of optimal speedup is $\frac{T_1}{CPL} = \frac{m}{2}$. For $n = 2^{14}$, the bound is 7. This upper bound is still too loose since communication is not included. This explains why the speedup does not scale up when p increases in Figure 7.15.

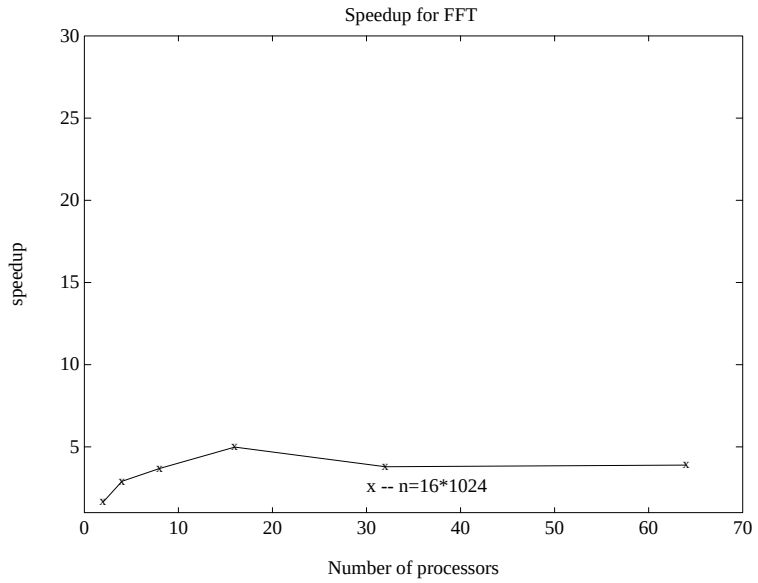


Figure 7.15: The speedup of FFT on nCUBE-II.

7.6 Sparse Matrix Computation

Since PYRROS can generate codes for irregular graphs, it can be used in applications where regular parallel programming is difficult or suffers from high overhead. An example is the solution of sparse matrix systems where dependence graphs are unstructured.

Finding an efficient scheduling solution for such kind of problem is important. An example is the area of circuit simulation and testing where the solution of a set of differential equations is often used. These problems are solved numerically by discretizing in time and space to reduce the problem to a large set of nonlinear equations. The solution is then derived by an iterative method such as Newton-Raphson which iterates over the same dataflow graph since the topology of the iteration matrix remains the same but the data change at each step, Karmarkar [56].

We have performed some tests on PYRROS performance for some sparse task graphs. Bhattacharjya [12] has developed a system that generates the sparse graphs using the GJ algorithm. It starts from a dense GJ task graph and deletes those tasks that perform operations on zeros. The test data come from Alvarado's sparse matrix manipulation system [4]. Figure 7.16 shows the dense graph that solves 8×8 matrix. and a sparse DAG after deleting useless tasks. We use PYRROS to schedule those

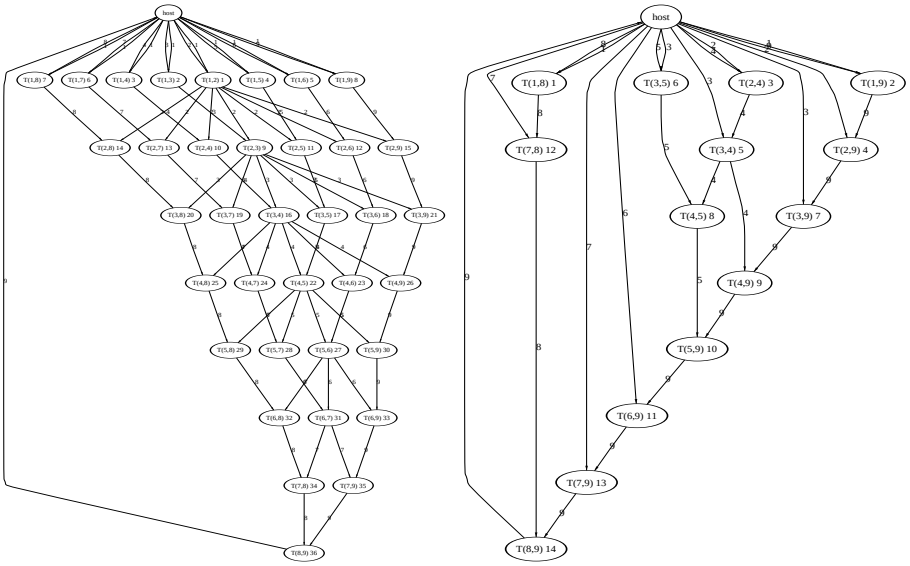


Figure 7.16: The left part is a dense matrix solving DAG. The right is a sparse DAG after deleting useless tasks.

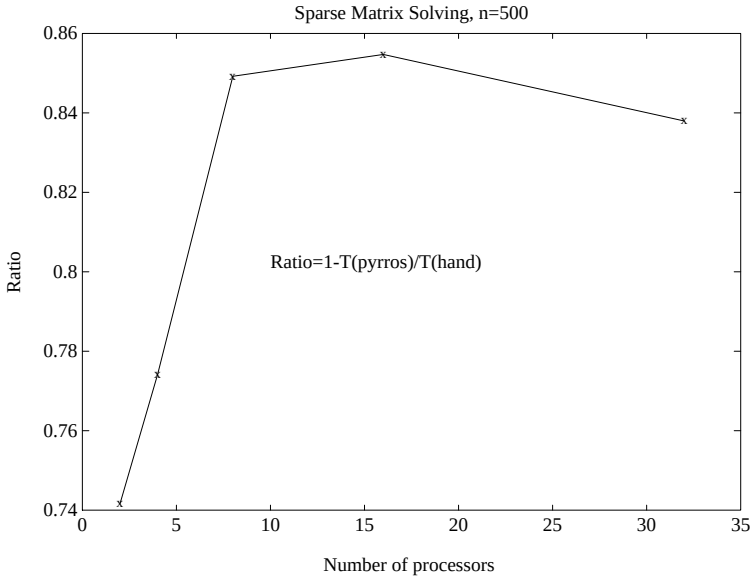


Figure 7.17: Dense with zero skipping vs. sparse dataflow graph on nCUBE-II.

sparse graphs and generate code for them. We have also written a program based on "owner computes rule" that uses a dense graph but skip operations on zeros. PYRROS achieves a large improvement compared with this hand-made program using regular scheduling. Figure 7.17 lists a comparison on nCUBE-II in solving a 500×500 sparse matrix system.

7.7 Partial-Differential Equations and Iterative Methods

In this section, we demonstrate the performance of iterative execution of a DAG schedule in parallelizing iterative numerical methods. In solving a partial-differential equation with boundary conditions, a region is discretized and an iterative method is used to approximate function values within this region. For example, Laplace partial-differential equation is

$$\frac{\partial^2 u}{\partial^2 x}(x,y) + \frac{\partial^2 u}{\partial^2 y}(x,y) = 0.$$

Figure 7.18 is an algorithm for deriving values of function u in a $n \times n$ mesh using the Gauss-Seidel iterative method.

```
for k = 1 to max_iter
  for i = 1 to n
    for j = 1 to n
      u_{i,j} = (u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1})/4;
    end
  end
end
```

Figure 7.18: Algorithm for Laplace partial-differential equation.

We first examine the task partitioning for the interior i and j loops. We partition the $n \times n$ mesh into $N \times N$ mesh blocks. Each block is a $r \times r$ submatrix where $n = N \times r$. The blocked algorithm with respect to loop i and j is shown in Figure 7.19. This partitioning is also known as tiling [96]. The dependence graph is shown in Figure 7.20(a) and (b).

We schedule this PDE DAG and repeatedly execute the same schedule with respect to loop k in Figure 7.18. Each processor executes assigned tasks one by one in one iteration. As soon as it finishes the last task, it starts a new iteration. The communication edges between two iterations are demonstrated in Figure 7.20. The pipelining of task

```
for bi = 1 to N
  for bj = 1 to N
T(bi, bj) : {
      for i = (bi − 1)r + 1 to (bi − 1)r + r
        for j = (bj − 1)r + 1 to (bj − 1)r + r
          u_{i,j} = (u_{i−1,j} + u_{i,j−1} + u_{i+1,j} + u_{i,j+1})/4;
        end
      end  }
  end
end
```

Figure 7.19: Partitioned algorithm using tiling for Laplace partial-differential equation.

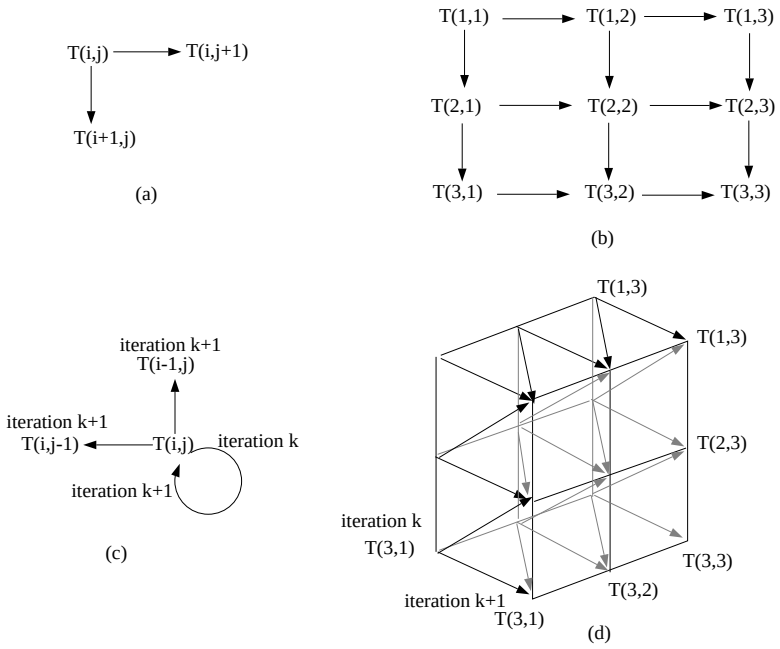


Figure 7.20: (a) Dependence pattern of the Laplace PDE DAG. (b) DAG when $N = 3$. (c) Dependence pattern between two iterations. (d) Communication between two iterations when $N = 3$.

execution between two iterations will improve the performance. The performance of PYRROS code for iteratively executing this DAG on nCUBE-II is shown in Figure 7.21. The number of iterations varies from 1 to 100. The maximum size of mesh solvable by PYRROS code on nCUBE-II with 4 MB per node is $6K \times 6K$ on 64 processors and $4K \times 4K$ on 32 and 64 processors with single precision arithmetic. We can see the performance improvement in terms of speedup when the number of iterations increases.

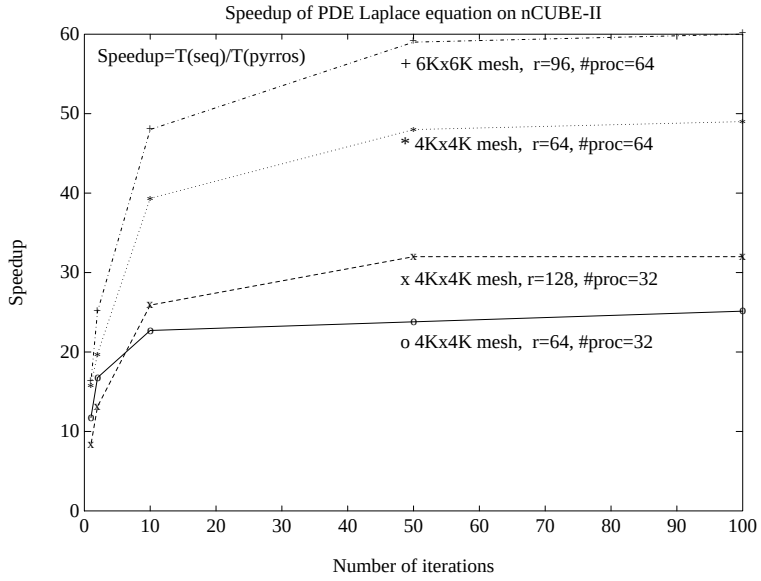


Figure 7.21: The performance of solving Laplace partial-different equation on nCUBE-II.

7.8 Conclusions on Experiments

The above experiments show that PYRROS code can attain a good performance if a proper task partitioning is provided. A user can execute partitioned programs on a parallel machine without involving the details in coding. Since exact weight estimation may not be feasible in practice, the experiments show that as long as the task graph is coarse grain and the weight assignment results in a coarse grain graph, the performance variations are small. Thus coarse grain partitions with sufficient parallelism are recommended. But even when coarse grain partitions cannot be derived, a compile time system such as PYRROS could still be useful in optimizing program performance. A user can utilize PYRROS to prototype and examine the performance of their programs

with different partitions and weight assignments.

Chapter 8

Related Work and Conclusions

8.1 Related Work

We describe several systems related to PYRROS and also present some comparison. We will also describe some future work.

SCHEDULER by Dongarra and Sorensen [30] uses centralized dynamic scheduling. This approach to scheduling and code generation is appropriate for shared memory architectures with few processors. For message passing architectures centralized scheduling will not perform well because of high control overhead in maintaining a global task queue.

PARAFRASE-2 [81] by Polychronopoulos et. al., is a parallelizing compiler system that performs dependence analysis, program partitioning and dynamic scheduling on shared memory machines. Recently they have proposed hierarchical task graph (HTG) for modeling parallel computation encapsulating both data and control dependence [47]. This representation provides a basis for exploiting both loop parallelism and unstructured functional parallelism. For message passing architectures, hierarchical granularity control is necessary to attain the performance and automatic scheduling for HTGs. This is still an open problem.

SISAL is a functional parallel language that allows users to explicitly express program parallelism. Sarkar [90] has developed an initial version of a compiler system that uses two step scheduling method for shared memory machines. We have used that idea and developed more efficient scheduling algorithms. Also we have developed a code generation method that integrates scheduling result for message passing architectures. Currently Feo, Cann and Oldehoeft [34] have been working on a new compiler system for SISAL.

The KALI parallel programming language by Koelbel and Mehrotra [61] has addressed code generation for DOALL parallelism. It does not address program scheduling. The DAG parallelism in PYRROS is more general. KALI system has been also focusing on run-time optimization with a similar idea as in the PARTI system by Saltz et.al [89]. When a problem is iterative in nature, its program dependence graph can be derived during the first iteration at run-time, and then data access information can be used to optimize communication for nonlocal data access in future iterations. The initial overhead of such run-time compilation is usually high but this cost is amortized over all iterations. PYRROS algorithms could be useful at the first iteration.

The CRYSTAL project by Chen, Choo and Li [16] has been targeted at developing a parallelizing compiler system for their functional language. Li and Chen [67] have recognized the importance of effective communication and have developed a communication routine selection system in CRYSTAL to generate code for message passing architectures. We have also proposed other communication optimizations and integrated them in one code generation scheme.

Fortran D project at Rice [51] and Fortran 90D at Syracuse [2] have emphasized compiling and code generation system for high performance Fortran. The mapping scheme is data driven, i.e. let a user define the data mapping and let the system determine the mapping of computation based on the owner compute rule. Our mapping scheme is computation driven, i.e., the assignment of computation first and data mapping is based on computation mapping. The SUPERB project by Zima et. al. [106] uses the similar techniques. Some recent research on data mapping by Gupta and Banerjee [49] has been automating data mapping process such that better load balancing can be achieved.

PREP-P by Berman [11] is a software system that automatically maps undirected communication graphs onto CHiP machine architectures. OREGAMI/LaRCS by Lo et. al. [68] software tool is used for the mapping of task graphs onto processors. Both systems use graph embedding algorithms to assign a set of tasks into processors. Precedence is not considered in PREP-P and OREGAMI/LaRCS.

HYPERTOOL by Wu and Gajski [98] and TaskGrapher by El-Rewini and Lewis [33]

have the same goals as PYRROS and use the same model of computation. TaskGrapher uses task duplication heuristics. Work by Chung and Ranka [19] also uses task duplication. The complexity of their algorithms is over $O(v^2)$.

HeNCE/PVM by Beguelin et. al [10] is a software system for exploiting large grain parallelism over heterogeneous computer networks. A graphic interface is provided for users to define program modules and their parallelism. The system integrates those computation modules and executes them over a network. CODE 2.0 by Newton and Browne [75] has a similar approach for coarse grain parallel computation. Scheduling is not fully considered yet in those systems and PYRROS algorithms could be useful.

8.2 Future Work

In this thesis, we have answered the following question: is it possible to automatically generate efficient scheduled code for message passing machines if dependence information is available? There are several directions for future work.

- *Applications and specialized tool systems*

  For many problems, for example, sparse matrix computation, dataflow graphs are totally irregular and automatic scheduling is very important since hand-made scheduling cannot perform well. Pozo [82] investigated the irregularity of sparse matrix graphs and showed that classical methods for shared memory machines do not work properly for message-passing architectures. It is mandatory to have a proper partitioning and automatic scheduling to gain scalable speedup. Thus for irregular problems, special scheduling and code generation systems can be developed for automating the optimization process.

  Other applications of PYRROS algorithms could be for coarse grain parallelism on a network of computers. Systems such as HeNCE/PVM [10] have been shown very useful in synthesizing program modules into a large software package. Scheduling techniques can be further used to improve the utilization of computer resources. Clustering techniques have been also shown useful for LINDA programming in [9].

- *Incorporating dependence analysis and program partitioning.*

We have seen that coarse grain parallelism is most successful for current message passing architectures. Automatic derivation of DAG parallelism is important for PYRROS. The general dependence analysis is NP-complete but for some special cases there exist polynomial algorithms [83, 95]. And systems such as ParaScope [58], PAT [7] and PTRAN [3] are helpful for users to uncover the parallelism. Cytron, Hind and Hsieh [28] presented an approach for DAG generation. Balasundaram and Kennedy [8] described a method to seek coarse grain parallelism by summarizing data access. Irigoin and Triolet [55], Wolf and Lam [96] addressed a program partitioning method for loop parallelism. We may use an interactive system that incorporate those methods and facilitate users to derive DAG computation.

- *General parallel computation*

  DAG computation still has its limitation. For many problems, task parallelism cannot be derived at compile time. Dynamic scheduling has been very successful for shared memory architectures to handle such problems [30, 80]. For distributed memory architectures, it is a challenging problem to balance run-time scheduling overhead and gain performance. If a problem is iterative, then the inspector and executor approach [61, 89] can be used. Our scheduling algorithms could be adapted to such a run-time optimization scheme.

  In general, a combination between static scheduling and dynamic scheduling is needed to assure both performance and flexibility on message passing distributed machines. For example, Ngai [76] used clustering techniques for run-time resource management. Also one could use static symbolic clustering but use dynamic scheduling within each processor at run-time.

  For loop parallelism, there is a regularity in task dependence. The size of an iteration space depends on some variables. Symbolic scheduling could provide solutions independent of the problem size, Quinton [84]. A DAG static scheduling algorithm can be used in evaluating various solutions of symbolic schedules.

On the other hand, control dependence needs to be considered for general computation. Girkar and Polychronopoulos [47]'s hierarchical task graph can be used for exploiting more general parallelism from a program. We plan to explore an approach that combines our methods to generate efficient and correct code for such graphs on message-passing architectures.

Appendix A

Specification of DAG Computation in PYRROS

In this appendix, we give a brief overview of PYRROS task graph language and graph specification for several DAGs using this language.

A.1 The Task Graph Language

The input of PYRROS system is a set of C programs with annotated dependence information. The task graph language has a C-like syntax and it provides a mechanism for users to write a DAG program that specifies the dependence relationships between C program modules.

Given a problem and an algorithm, a user first partitions this algorithm and derives a directed acyclic task graph. Then he writes C code to specify task operations and the data structures accessed by those tasks. Also he needs to write how the host program invokes the computation of this DAG. A DAG program is used to specify tasks, data, host procedures, and their relationship. PYRROS can synthesize those C modules and generate parallel code C modules based on DAG specification.

The first part of a DAG program contains the parameters and files that contain low-level C procedure definitions. Function *set_strufile*() gives the names of .h files. *set_nodefile*() gives the names of files containing C procedures used in task computation. *set_hostfile*() gives the names of files containing C procedures used in host program. *set_processor*() sets up the architecture parameters.

The second part is the specification of data structures accessed by tasks. We use C struct to define data to be accessed during DAG computation.

The third part is a generic definition of tasks with respect to their indices. Statement *taskdim* defines the index space of tasks to be used in graph specification. Statement

task defines a task. *set_weight*() gives an expression for task computation weight assignment. *read*() and *write*() specify the incoming and outgoing dependence edges. The first argument of *read*() is the name of a data item to be used and the second is the size of the data item in bytes. The first argument of *write*() is the data name, the second argument is the name of task that uses that data item, and the third is the data size .

The fourth part is a specification of this DAG. We can use *for* and other control statements to specify iterative patterns. *eval_task*() gives the name of a task instance to be used.

The fifth part is a specification of the host program. If there is some data to be sent from host to tasks in this DAG or to be received from tasks, function *read*() and *write*() can be used. *eval_dag*() is put in between those *write*() and *read*() to formally invoke DAG computation.

For the iterative execution of a schedule for a DAG, *iread*() and *iwrite*() specify data communicated between iterations. *iwrite*() sends data items to a task in the next iteration if the current iteration is not the last iteration. *iwriteL*() is the same as *iwrite*() except that it sends data items to the host at the last iteration. *iread*() receives data items from a task in the previous iteration if the current iteration is not the first iteration. *ireadF*() is the same as *iread*() except that it reads data items from the host at the first iteration.

A.2 GJ

GJ without pivoting

```
/*GJ without pivoting parameter specification*/


#define NoProc 16
#define alpha 200
#define beta 0.6
#define w 2.3
```

```
#define n 50 /*Number of block columns*/
#define bsize1 4
#define bsize2 bsize1*bsize1
set_strufile( "block.h"); /*h file */
set_hostfile("block_host.c"); /*c functions for host*/
set_nodefile ("blocknodesup.c"); /*c function update */
set_processor(NoProc,alpha,beta, w);
/*data specification*/
struct Dataitem colbk[n+1];
/*task specification*/
taskdim T[n][n+1];
task T[k][j]{
    int bc;
    set_weight(bsize2*bsize1*n);
    read(&colbk[k], bsize2*n*4);
    read(&colbk[j], bsize2*n*4);
    c_update(&colbk[k], &colbk[j],k,j);
    if(k<n-1)
       if(k != j-1)
        write(&colbk[j], T[k+1][j], bsize2*n*4);
       else for(bc=j+1;bc<=n; bc=bc+1)
            write(&colbk[j], T[k+1][bc], bsize2*n*4);
    else
            write(&colbk[j], host, bsize2*n*4);
}
/*graph specification*/
dag gjdag{
    int k;
    int j;
    for(k=0; k<n; k=k+1)
```

```
        for(j=k+1; j<=n; j=j+1){
            eval_task(T[k][j]);
        }
}
/*host program specification*/
program{
    int j;
    init_data(colbk,n);
    data_swap(colbk,n);
    for(j=1; j<=n; j=j+1){
        write(&colbk[0],T[0][j],bsize2*n*4);
        write(&colbk[j],T[0][j],bsize2*n*4);
    }
    eval_dag(gjdag);
    read(&colbk[n],bsize2*n*4);
    data_swap1(&colbk[n],n);
    post_op(colbk,n);
}
```

GJ with pivoting

```
/* Solve Ax=b using GJ with pivoting*/
#define n 2000 /*block column numbers*/
#define N 125
#define bs 16
#define bs2 bs*bs
#define ES 4
#define ES2 2
set_strufile( "newgjp.h"); /*h file */
set_hostfile("gjp_host.c"); /*c functions for host*/
set_nodefile ("gjp_node.c"); /*c function pivot, update.*/
```

```
set_processor(64,160,0.6, 2.38);
struct bcol colbk[N];
struct col b;
struct vect piv_rec;
taskdim T[N][N];/*updating matrix tasks*/
taskdim P[N]; /*pivoting tasks*/
taskdim B[N]; /*updating  column b */
task P[k]{
    int bc;
    set_weight(bs*(bs+1)*n/2);
    set_cluster(k);
    if(k>0){
        read(&colbk[k], bs*(n+1)*ES);
        read(&piv_rec, (n+1)*ES2);
    }
    bpivot(&colbk[k], k, &piv_rec);
    write(&colbk[k], B[k], (n+1)*bs*ES);
    if(k<N-1){
       for(bc=k+1;bc<N; bc=bc+1)
            write(&colbk[k], T[k][bc], (n+1)*bs*ES);
       write(&piv_rec, P[k+1], (n+1)*ES2);
    }else write(&piv_rec, host, (n+1)*ES2);
}
task T[k][j]{
    set_weight(bs2*n);
    set_cluster(j-1);
    read(&colbk[k], (n+1)*bs*ES);
    if(k>0){
        read(&colbk[j], (n+1)*bs*ES);
    }
```

```
    update_mat(k,j,&colbk[k], &colbk[j]);
    if(j==k+1)
        write(&colbk[k+1], P[k+1], (n+1)*bs*ES);
    else
        write(&colbk[j], T[k+1][j], (n+1)*bs*ES);
}
task B[k]{
    set_weight(bs*n);
    set_cluster(N);
    read(&colbk[k], (n+1)*bs*ES);
    if(k>0){
        read(&b, (n+1)*ES);
    }
    update_b(k,&colbk[k], &b);
    if(k<N-1)
        write(&b, B[k+1], (n+1)*ES);
    else
        write(&b, host, (n+1)*ES);
}

dag gjpdag{
    int k; int j;
    for(k=0; k<N; k=k+1){
        eval_task(P[k]);
        for(j=k+1; j<N; j=j+1){
            eval_task(T[k][j]);
        }
        eval_task(B[k]);
    }
}
```

```
program{
    eval_dag(gjpdag);
    read(&piv_rec,(n+1)*ES2);
    read(&b,(n+1)*ES);
    data_swap1(&b,&piv_rec,n);
    print_result(&b,n, &piv_rec );
}
```

A.3 LU

LU without pivoting

```
#define NoPROC 8
#define Alpha 200
#define Beta 0.6
#define w 2.3
#define n 100 /*Number of block columns*/
#define bs1 2
#define bs2 bs1*bs1
#define bs3 bs2*bs1
set_strufile( "block.h");        /*h file */
set_hostfile("block_host.c");    /*c functions for host*/
set_nodefile ("blocknodesup.c");    /*c function update and taskkk*/
set_processor(NoPROC,Alpha,Beta, w);
struct Dataitem colbk[n];
struct Dataitem partcolbk[n];
taskdim P[n];
taskdim T[n][n];
task P[k]{
    int bc;
    set_weight((n-k)/2*bs3);
```

```
    read(&colbk[k], n*4*bs2);
    task_kk(&colbk[k], k, &partcolbk[k]);
    for(bc=k+1;bc<n; bc=bc+1)
        write(&partcolbk[k], T[k][bc], (n-k)*4*bs2);
    write(&colbk[k], host, n*4*bs2);
}
task T[k][j]{
    int bc;
    set_weight((n-k)*bs3);
    read(&partcolbk[k], (n-k)*4*bs2);
    read(&colbk[j], n*4*bs2);
    c_update(&partcolbk[k], &colbk[j],k,j);
    if(j==k+1)
        write(&colbk[j], P[k+1], n*4*bs2);
    else write(&colbk[j], T[k+1][j], n*4*bs2);
}
dag lubdag{
    int k; int j;
    for(k=0; k<n; k=k+1){
        eval_task(P[k]);
        for(j=k+1; j<n; j=j+1)
            eval_task(T[k][j]);
    }
}
program{
    int j;
    init_data(colbk,n);
    data_swap(colbk,n);
    write(&colbk[0],P[0],n*4*bs2);
    for(j=1; j<n; j=j+1) write(&colbk[j],T[0][j],n*4*bs2);
```

```
    eval_dag(lubdag);
    for(j=0; j<n; j=j+1) read(&colbk[j],n*4*bs2);
    data_swap(colbk,n);
    post_op(colbk,n);
}
```

LU with pivoting

```
#define NDIM 16 /*matrix dim */
#define N 8 /*block colbkumn numbers*/
#define NR 2/*block size */
#define bs2 NR*NR
#define bs3 bs2*NR
#define ES 4
#define ES2 2
set_strufile( "lup.h"); /*h file */
set_nodefile ("lup_node.c"); /*c function update and pivoting*/
set_processor(4,200,0.6, 2.34);
struct Dataitem colbk[N];
struct Dataitem1 pivcolbk[N];
struct Dataitem2 piv_rec;
struct Sig stop;
taskdim P[N];
taskdim T[N][N];
task P[k]{
    int bc;
    set_weight((N-k+1)*bs3);
    if(k>0){
         read(&colbk[k], NDIM*NR*ES);
         read(&piv_rec, NDIM*ES2);
    }
```

```
    bpivot(&colbk[k], k, &piv_rec, &pivcolbk[k]);
    for(bc=k+1;bc<N; bc=bc+1)
        write(&pivcolbk[k], T[k][bc], NDIM*ES2+NDIM*NR*ES);
    if(k<N-1)
        write(&piv_rec, P[k+1], NDIM*ES2);
    else
        write(&stop, host, ES2);
}
task T[k][j]{
    set_weight((N-k+1)*bs3);
    read(&pivcolbk[k], NDIM*ES2+NDIM*NR*ES);
    if(k>0) read(&colbk[j], NDIM*NR*ES);
    update_mat(k,j,&pivcolbk[k], &colbk[j]);
    if(j==k+1)
        write(&colbk[j], P[k+1], NDIM*NR*ES);
    else write(&colbk[j], T[k+1][j],  NDIM*NR*ES);
}
dag lupbdag{
    int k; int j;
    for(k=0; k<N; k=k+1){
        eval_task(P[k]);
        for(j=k+1; j<N; j=j+1)
            eval_task(T[k][j]);
    }
}
program{
    eval_dag(lupbdag);
    read(&stop, ES2);
}
```

A.4 Matrix Multiplication

```
/*paraemeter specification*/
#define NoPROC 4
set_strufile("matrix.h");
set_hostfile("matrix_host.c");
set_nodefile("matrix_node.c");
set_processor(NoPROC,160, 0.6, 1.6);
/*data specification*/
#define n 100
struct Mat b;
struct Row x[n];
/*task specification*/
taskdim  Y;
task Y {
    int j;
    set_weight(n*n/2);
    init_matrix(&b,n);
    for(j=0;j<n;j=j+1) write(&b,T[j],n*n*4);
}
taskdim  T[n];
task T[i] {
    set_weight(n*n);
    read(&b, n*n*4);
    mat_vect(&b,n,&x[i]);
    write(&x[i],host,4*n);
}
dag matrix {
    int i;
    eval_task(Y);
```

```
    for(i=0;i<n;i=i+1){
        eval_task(T[i]);
    }
}
program{
    int i;
    eval_dag(matrix);
    for(i=0;i<n;i=i+1){
        read(&x[i],4*n);
    }
    dodataswap(x,n);
    print_matrix(x,n);
}
```

A.5 FFT

```
/*paraemeter specification*/
#define NoPROC 4
set_strufile("fft.h");
set_nodefile("fft_node.c");
set_processor(NoPROC,200, 0.6, 1.6);
/*k is to control the finest level, k <= log n */
#define n 1024
#define k 5
/*data specification*/
struct vect x[k+1][n];
struct mysig stop;
/*task specification*/
taskdim  SPLIT[k][n/2];
task SPLIT[i][j] {
    set_weight(0.5*n/(2^i));
```

```
    if(i>0)
        read(&x[i][j], 8*n/ (2^i));
    split_data(i,j,&x[i][j], n/(2^i), &x[i+1][2*j], &x[i+1][2*j+1]);
    if(i< k-1){
         write(&x[i+1][2*j],SPLIT[i+1][2*j], 8*n/(2^(i+1)));
         write(&x[i+1][2*j+1],SPLIT[i+1][2*j+1], 8*n/(2^(i+1)));
    }else{
         write(&x[i+1][2*j],MERGE[i][j], 8*n/(2^(i+1)));
         write(&x[i+1][2*j+1],MERGE[i][j], 8*n/(2^(i+1)));
    }
}
taskdim  MERGE[k][n/2];
task MERGE[i][j] {
    set_weight(2*n/(2^i));
    read(&x[i+1][2*j],8*n/(2^(i+1)));
    read(&x[i+1][2*j+1],8*n/(2^(i+1)));
    merge_data(i,j,&x[i+1][2*j], &x[i+1][2*j+1], &x[i][j],n/(2^i));
    if(i==0) write(&stop, host,4);
    else
        write(&x[i][j], MERGE[i-1][j/2],8*n/(2^i));
}

dag fft {
    int i;
    int j;
    for(i=0;i<k;i=i+1){
        for(j=0;j< 2^i;j=j+1){
            eval_task(SPLIT[i][j]);
        }
    }
```

```
    for(i=k-1;i>=0;i=i-1)
        for(j=0;j< 2^i;j=j+1)
            eval_task(MERGE[i][j]);
}

program{
    eval_dag(fft);
    read(&stop,4);
}
```

A.6 PDE

```
/*initial data is sent from host,
a[i][j] is updated after each iteration*/
/*paraemeter specification for pde task*/
set_strufile("pde.h");
set_hostfile("pde_h.c");
set_nodefile("pde_n.c");
set_processor(8, 200, 0.6, 4);
#define n 32
#define bs 64
#define bs2 (bs+2)*(bs+2)
#define ES 8
/*data specification*/
struct block a[n][n]; /* nxn block, each block of size bs+2, extended*/
struct blockbound southb[n][n];
struct blockbound eastb[n][n];
struct blockbound northb[n][n];
struct blockbound westb[n][n];
/*task specification*/
taskdim  T[n][n];
```

```
task T[i][j] {
    set_weight(bs*bs);
    set_cluster(j);
    ireadF(&a[i][j],ES*bs2);/*reading including the first*/
    if(i>0) read(&southb[i-1][j],ES*bs);
    if(j>0) read(&eastb[i][j-1],ES*bs);
    if(i<n-1) iread(&northb[i+1][j],ES*bs);
    if(j<n-1) iread(&westb[i][j+1],ES*bs);
    blockpde(i,j,&a[i][j],
    &southb[i-1][j], &eastb[i][j-1], &northb[i+1][j], &westb[i][j+1],
        &southb[i][j],&eastb[i][j], &northb[i][j],&westb[i][j]);
    if(i<n-1) write(&southb[i][j],T[i+1][j],ES*bs);
    if(j<n-1) write(&eastb[i][j],T[i][j+1],ES*bs);
    if(i>0) iwrite(&northb[i][j],T[i-1][j], ES*bs);
    if(j>0) iwrite(&westb[i][j],T[i][j-1], ES*bs);
    iwriteL(&a[i][j],T[i][j],ES*bs2);
}
dag pdedag{
    int i; int j;
    for(i=0;i<n;i=i+1)
        for(j=0;j<n;j=j+1){
            eval_task(T[i][j]);
    }
}
program {
    int i; int j;
    init_data(a, n,bs);
    dodataswap(a,n,bs);
    for(i=0;i<n;i=i+1){
        for(j=0;j<n;j=j+1){
```

```
            write(&a[i][j], T[i][j], ES*bs2);
        }
    }
    eval_dag(pdedag,10);
    for(i=0;i<n;i=i+1)
        for(j=0;j<n;j=j+1){
            read(&a[i][j],ES*bs2);
        }
    dodataswap(a,n,bs);
    print_matrix(a,n,bs);
}
```

References

[1] T. Adam, K.M. Chandy and J. R. Dickson, A comparison of list schedules for parallel processing Systems, *CACM*, 17:12 (1974), 685-690.

[2] I. Ahmad, A. Choudhary, G. Fox, K. Parasuram, R. Ponnusamy, S. Ranka, and R. Thakur, Implementation and scalability of Fortran 90D intrinsic functions on distributed memory machines, NPAC Tech Report, Syracuse University, 1993.

[3] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, An overview of the PTRAN analysis system for multiprocessing, *J. of Parallel and Dist Computing*, Oct 88.

[4] F. L. Alvarado, The sparse matrix manipulation system users manual, University of Wisconsin, Oct. 1990.

[5] M. A. Al-Mouhamed, *Lower Bound on the Number of Processors and Time for Scheduling Precedence Graphs with Communication Costs*, IEEE Trans. on Software Engineering, vol. 16, no. 12, pp. 1390-1401, 1990.

[6] F.D. Anger, J. Hwang, and Y. Chow, Scheduling with sufficient loosely coupled processors, *J. of Parallel and Distributed Computing*, vol. 9, pp. 87-92, 1990.

[7] B. Appelbe, C. McDowell, and K. Smith, Start/Pat: A parallel programming toolkit, *IEEE Software* 6, 4 (July 1989), pp. 29-38.

[8] V. Balasundaram and K. Kennedy, A technique for summarizing data access and its use in parallelism enhancing transformations, *Proc. of 1989 ACM SIGPLAN*, pp. 41-53.

[9] A. Bakre and A. Gerasoulis, Numerical applications in LINDA on message-passing architectures, Report, 1992.

[10] A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek, V.S. Sunderam, Graphical development tools for network-based concurrent supercomputing, *Proc. of Supercomputing '91*, IEEE, 1991, pp.435-444.

[11] F. Berman, Experience with an automatic solution to the mapping problem. In L. Jamieson, D. Gannon an R. Douglass, Eds., *The Characteristics of Parallel Algorithms* (MIT press, 1987) 307-334.

[12] P. Bhattacharjya, Gauss-Jordan elimination for sparse matrix on message-passing architecture, Report, 1991.

[13] S. Bokhari, On the mapping problem, *IEEE Trans. Comput.* C-30 3(1981), 207-214.

[14] D. Callahan, and K. Kennedy, Compiling Programs for Distributed-memory Multiprocessors, *Journal of Supercomputing*, Vol. 2, 1988, pp. 151-169.

[15] D. Cann, Retire Fortran? A Debate Rekindled, *Proc. of Supercomputing 91*, IEEE, pp 264-272, 1991.

[16] M. Chen, Y.I Choo and J. Li, Compiling Parallel Programs by Optimizing Performance, *J. of Supercomputing*, 2, 171-207 (1988).

[17] D. Y. Cheng and D. M. Pase, An Evaluation of Automatic and Interactive Parallel Programming Tools, *Proc. of IEEE Supercomputing 91*, pp. 412-423.

[18] H. Cheong and A. V. Veidenbaum, Stale data detection and coherence enforcement using flow analysis, *Proc. of Intel. Conf. on Parallel Processing*, 1988, pp. 138-145.

[19] Y. C. Chung and S. Ranka, An optimization approach for static scheduling of directed-acyclic graphs on distributed memory multiprocessors, Report, Syracuse Univ, 1992.

[20] Ph. Chretienne, Task Scheduling over Distributed Memory Machines, *Proc. of Inter. Workshop on Parallel and Distributed Algorithms*, (North Holland, Ed.), 1989.

[21] Ph. Chretienne, A Polynomial Algorithm to Optimially Schedule Tasks over an ideal Distributed System under Tree-like Presedence Constraints, *European Journal of Operational Research*, 2:43 (1989), pp225-230.

[22] Ph. Chretienne, Complexity of Tree Scheduling with Interprocessor Communication Delays, Tech. Report, M.A.S.I. 90.5, Universite Pierre et Marie Curie, 1990.

[23] E.G. Coffman and R.L. Graham, Optimal scheduling for two-processors systems, *Acta Informatica*, 3 (1972), 200-213.

[24] J. Y. Colin and Ph. Chretienne, C.P.M. Scheduling with Small Communication Delays and Task Duplication, Report, M.A.S.I. 90.1, Universite Pierre et Marie Curie, 1990.

[25] J. W. Cooley and J. W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation*, Vol. 19, No. 90, 1965, pp. 297-301.

[26] M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystram, Parallel Gaussian Elimination on an MIMD Computer, *Parallel Computing,* vol. 6, pp. 275-296, 1988.

[27] R. Cytron and J. Ferrante, What's in a name? The Value of Renaming for Parallelism Detection and Storage Allocation, *Proc. of ICPP 87*, pp. 19-27.

[28] R. Cytron, M. Hind, and W. Hsieh, Automatic generation of DAG parallelism, *Proc. of SIGPLAN Conf. on Programming Language Design and Implementations*, 1989, pp. 54-68.

[29] M. Cosnard, B. Tourancheau, and G. Villard, Gaussian Elimination on Message Passing Architecture, *Lecture Notes in Computer Science 297*, Berlin: Springer-Verlag, 1987, pp. 611-628.

[30] J.J. Dongarra, . and D. C. Sorensen, SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs, in *The Characteristics of Parallel Algorithms,* D.B. Gannon, L.H. Jamieson and R.J. Douglass (Eds), MIT Press, 1987, pp363-394.

[31] J.J. Dongarra, I. Duff, D.C. Sorensen, and H.A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers,* SIAM press,1991.

[32] T.H. Dunigan, Performance of the INTEL iPSC/860 and nCUBE 6400 hypercube, ORNL/TM-11790, Oak Ridge National Lab., TN, 1991.

[33] H. El-Rewini and T.G. Lewis, Scheduling parallel program tasks onto arbitrary target machines, *J. of Parallel and Distributed Computing*, 9(1990), 138-153.

[34] J. Feo, D. Cann, and R. Oldehoeft, A report on the Sisal language project, *J. of Parallel and Distributed Computing*, 10(1990), pp. 349-365.

[35] J. Ferrante, K. Ottenstein, and J. Warren, The program dependence graph and its use in optimization, *ACM Trans. on Programming languages and Systems*, Vol. 9, No. 3, 1987, pp. 319-349.

[36] M. R. Garey and D.S. Johnson, Computers and Intractability, *a Guide to the Theory of NP-completeness*, W.H. Freeman and Company: New York, 1979.

[37] G.A. Geist and M.T. Heath, Matrix Factorization on a Hypercube Multiprocessor, in *Hypercube Multiprocessors,* Philadelphia:SIAM, 1986, pp. 161-180.

[38] A. George, , M.T. Heath, and J. Liu, Parallel Cholesky factorization on a shared memory processor, *Lin. Algebra Appl.,* 77(1986), pp. 165-187.

[39] A. Gerasoulis, I. Nelken. Static scheduling for linear algebra DAGs. *Proc. of 4th Conf. on Hypercubes*, Monterey, Vol. 1, 1989, 671-674.

[40] A. Gerasoulis , S. Venugopal, and T. Yang, Clustering task graphs for message passing architectures, *1990 Proc. of ACM Inter. Conf. on Supercomputing,* Amsterdam, pp. 447-456, 1990.

[41] A. Gerasoulis and S. Venugopal, Linear clustering of linear algebra task graphs for local memory systems, Report, 1990.

[42] A. Gerasoulis and T. Yang, On the granularity and clustering of directed acyclic task graphs, To appear in *IEEE Trans. on Parallel and Distributed Systems.*, 1993.

[43] A. Gerasoulis and T. Yang, Static scheduling of parallel programs for message passing architectures.*Lecture Notes in Computer Science, No. 634, Parallel Processing: CONPAR 92 – VAPP V*, Springer-Varlag, 1992, pp. 601-612.

[44] A. Gerasoulis and T. Yang, A comparison of clustering heuristics for scheduling DAGs on multiprocessors, *J. of Distributed and Parallel Computing,* special issue on scheduling and load balancing, Vol. 16, No. 4, pp. 276-291 (Dec. 1992).

[45] A. Gerasoulis and T. Yang, Scheduling program task graphs on MIMD architectures, To appear as a book chapter in *Algorithm Derivation and Program Transformation*, R. Paige, J. Reif, and R. Wachter (Eds.), Kluwer Publisher, 1992.

[46] M. Girkar and C. Polychronopoulos, Partitioning programs for parallel execution, *Proc. of the 1988 ACM Inter. Conf. on Supercomputing,* St. Malo, France, July 4-8, 1988.

[47] M. Girkar and C. Polychronopoulos, Automatic extraction of functinal parallelism from ordinary programs, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 3, No. 2, 1992, pp. 166-178.

[48] R. L. Graham, Bounds for certain multiprocessing anomalies, *Bell System Tech. J.* , 45 (1966), 1563-1581.

[49] M. Gupta and P. Banerjee, Demonstration of automatic data partitioning techniques for parallelizing compilers on Multicomputers, *IEEE. Trans. on Parallel and Distributed Systems*, Vol 3, No. 2, Mar. 1992, pp. 179-193.

[50] M.S. Hecht, *Flow Analysis of Computer Programs*, North-Holland, New York, 1977.

[51] S. Hiranandani, K. Kennedy, and C.W. Tseng, Compiler optimizations for Fortran D on MIMD distributed-memory machines, *Proc. of Supercomputing '91*, IEEE, pp. 86-100.

[52] J.A. Hoogeveen, S.L. van de Velde, and B. Veltman, Complexity of scheduling multiprocessor tasks with prespecified processor allocations, CWI, Report BS-R9211 June 1992, Netherlands.

[53] T.C. Hu, Parallel sequencing and assembly lines problems, *Operations Research*, 9 (1961), 841-848.

[54] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, Scheduling precedence graphs in systems with interprocessor communication times, *SIAM J. Comput.,* pp. 244-257, 1989.

[55] F. Irigoin and R. Triolet, Supernode partitioning, *Proc. of ACM SIGPLAN Symposi. on Principles of Programming Languages*, 1988, pp. 319-329.

[56] N. Karmarkar, A new parallel architecture for sparse matrix computation based on finite project geometries, *Proc. of Supercomputing '91*, IEEE, pp. 358-369.

[57] H. Kasahara and S. Narita. Practical multi-processor scheduling algorithms for efficient parallel processing, *IEEE Trans. on Computers,* C-33 (1984), pp. 1023-1029.

[58] K. Kennedy, K.S. NcKinley, and C. Tseng, Analysis and transformation in the ParaScope Editor, In *Proc. of 1991 ACM Inter. Conf. on Supercomputing*, Germany, 1991.

[59] S. J. Kim, A general approach to multiprocessor scheduling, TR-88-04, DCS, Univ. of Texas at Austin, 1988.

[60] S.J. Kim and J.C Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, *Proc. of Int'l Conf. on Parallel Processing*, vol 3, pp. 1-8, 1988.

[61] C. Koelbel, and P. Mehrotra, Compiling global name-space parallel loops for distributed exection, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 2, No. 4 , 1991, pp. 440-451.

[62] B. Kruatrachue B. and T. Lewis, Grain Size Determination for Parallel Processing, *IEEE Software*, pp. 23-32, Jan. 1988.

[63] S.Y. Kung, *VLSI Array Processors*, New Jersey:Prentice Hall, 1988.

[64] P. Ladkin and B. Simons, Compile-time analysis of communicating processes, *Proc. of 6th ACM Inter. Conf. on Supercomputing,* Washington D.C., July, 1992, pp. 248-259.

[65] Y. Lan, A. H. Esfahanian, and L. M. Ni, Multicast in hypercube multiprocessors, *Journal of Parallel and Distributed Computing*, 8, 30-41 (1990).

[66] J. K. Lenstra and A.H.G. Rinnooy Kan, Complexity of Scheduling under Precedence Constraints, *Operation Research*, 26:1 (1978).

[67] J. Li and M. Chen, Compiling Communication-Efficient Programs for Massively Parallel Machines, *IEEE Trans. on Parallel and Distributed Systems*, Vol. 2, No. 3 , 1991, pp. 361-376.

[68] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. Mohamed, J. A. Telle, OREGAMI: software tools for mapping parallel computations to parallel architectures, *Proc. Int'l. Conf. on Parallel Processing*, 1990, Vol. II 88-92.

[69] R.E. Lord, J.S. Kowalik, and S. P. Kumar, Solving linear algebraic equations on an MIMD computer, *Journal of the ACM*, vol. 30, pp. 103-117, 1983.

[70] P. K. McKinley, H. Xu, A. H. Esfahanian, and L. M. Ni, Unicast-based multicast communication in wormhole-routed networks, *Proc. of 1992 Inter. Conf. on Parallel Processing*, Vol. II, pp. 10-19.

[71] S. Masticola and B. G. Ryder, Static infinite wait anomaly detection in polynomial time, *Proc. of Inter. Conf. on Parallel Processing*, 1990, pp II78-II87.

[72] C. McGreary C. and H. Gill H, Automatic determination of grain size for efficient parallel processing, *Communications of ACM,* vol. 32, pp. 1073-1078, Sept., 1989.

[73] C. Moler, Matrix Computation on Distributed Memory Multiprocessors, In *Hypercube Multiprocessors 1986*, SIAM, pp. 181-195.

[74] I. H. Nelken, Parallelization for MIMD multiprocessors with applications to linear algebra algorithms, Ph.D Thesis, LCSR-TR-134, Rutgers University, 1989.

[75] P. Newton and J. Browne, The CODE 2.0 graphical parallel programming language, *Proc. of 6th ACM Inter. Conf. on Supercomputing,* Washington D.C., July, 1992, pp. 167-177.

[76] T. F. Ngai, Runtime resource management in concurrent systems, Tech. Report, CSL-TR-92-504, Stanford Univ., Jan. 1992.

[77] J.M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, New York:Plenum, 1988.

[78] C. Papadimitriou and M. Yannakakis, Towards on an architecture-independent analysis of parallel algorithms, *SIAM J. Comput.,* vol. 19, pp. 322-328, 1990.

[79] Picouleau, C., Two new NP-Complete scheduling problems with communication delays and unlimited number of processors, M.A.S.I, Universite Pierre et Marie Curie Tour 45-46 B314, 4, place Jussieu, 1991.

[80] C. D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[81] C. D. Polychronopoulos, M. Girkar, M. Haghighat,C. Lee, B. Leung, and D. Schouten, The Structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran, in *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau and D. Padua (Eds.), 1990.

[82] R. Pozo, Performance modeling of sparse matrix methods for distributed memory architectures, in *Lecture Notes in Computer Science, No. 634, Parallel Processing: CONPAR 92 – VAPP V*, Springer-Varlag, 1992, pp. 677-688.

[83] W. Pugh, The Omega test: a fast and practical integer programming algorithm for dependence analysis, *Proc. of Supercomputing '91*, IEEE, NM, 1991, pp. 4-13.

[84] P. Quinton, Mapping recurrences on parallel architectures, *Proc. of Int. Conf on Supercomputing ICS 88*, L.P. Kartashev and S.I. Kartashev Eds., 1988, III 1-8.

[85] Y. Robert, B. Tourancheu, and G. Villard, Data allocation strategies for the Gauss and Jordan algorithms on a ring of processors, *Information Processing Letters*, vol. 31, pp.21-29, 1989.

[86] A. Rogers and K. Pingali, Process decomposition through locality of reference, *Proc. of SIGPLAN 1989*, pp. 69-80.

[87] Y. Saad, Gaussian elimination on hypercubes, in *Parallel Algorithms and Architectures*, Cosnard, M. et al. Eds., Elsevier Science Publishers, North-Holland, 1986.

[88] Y. Saad and M. H. Schultz, Data communication in hypercubes, DCS/RR-428, Research Report, Yale University, 1985.

[89] Saltz, J., Crowley, K., Mirchandaney, R. and Berryman,H., Run-time scheduling and execution of loops on message passing machines, *J. of Parallel and Distributed Computing,* Vol. 8, 1990, pp. 303-312.

[90] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, The MIT Press, 1989.

[91] V. Sarkar, Determining average program execution times and their variance, *Proc. of 1989 SIGPLAN*, ACM, pp. 298-312.

[92] D. Shasha and M. Snir, Efficient and correct execution of parallel programs that share memory, *ACM Trans. on Programming Languages and Systems*, Vol 10, No. 2, 1988, pp. 282-312.

[93] H. Stone, *High-Performance Computer Architectures*, Reading, Mass.: Addison-Wesley, 1987.

[94] P. Tseng, Compiling programs for a linear systolic array, *Proc. of SIGPLAN 1990*, pp 311-321.

[95] M. Wolfe and U. Banerjee, Data dependence and its application to parallel processing, *International Journal of Parallel Programming,* Vol. 16, No. 2, 1987, pp. 137-178.

[96] M. E. Wolf and M. S. Lam, A loop transformation theory and an algorithm to maximize parallelism, *IEEE Transactions on Parallel and Distributed Systems,* Oct, 1991, pp. 452-471.

[97] R. Wolski and J. Feo, *Program Parititoning for NUMA Multiprocessor Computer Systems*, Tech. Report, Lawrence Livermore Nat. Lab., 1992.

[98] M. Y. Wu and D. Gajski, Hypertool: A programming aid for message-passing systems, *IEEE Trans. on Parallel and Distributed Systems,* vol. 1, no. 3, pp.330-343, 1990.

[99] M. Y. Wu, Personal Communications, Feb. 1993.

[100] J. Yang, L. Bic and A. Nicolau, A mapping strategy for MIMD computers, *Proc. of 1991 Inter. Conf. on Parallel Processing*, vol I, pp. 102-109.

[101] T. Yang and A. Gerasoulis, A fast static scheduling algorithm for DAGs on an unbounded number of processors, *Proc. of Supercomputing '91,* IEEE, Albuquerque, NM, 1991, pp. 633-642.

[102] T. Yang and A. Gerasoulis. A parallel programming tool for scheduling on distributed memory multiprocessors. *Proc. of Scalable High Performance Computing Conference*, IEEE, Williamsburg, VA., April, 1992, pp. 350-357.

[103] T. Yang and A. Gerasoulis, PYRROS: Static task scheduling and code generation for message-passing multiprocessors, *Proc. of 6th ACM Inter. Conf. on Supercomputing,* Washington D.C., July, 1992, pp. 428-437.

[104] T. Yang and A. Gerasoulis, List scheduling with and without communication delay, To appear in *Parallel Computing*, 1993.

[105] T. Yang and A. Gerasoulis, DSC: Scheduling parallel tasks on an unbounded number of processors, Submitted for publication.

[106] H. Zima, H. J. Bast, and M. Gerndt, SUPERB: A tool for semi-automatic MIMD/SIMD parallelization, *Parallel Computing*, Vol 6., 1988, pp. 1-18.

Vita

Tao Yang

1980 Graduated from Zhou Shan High School, Zhou Shang, China.

1980-84 Undergraduate in Zhejiang University, Hangzhou, China. B.S in Computer Science.

1984-87 Graduate work in Zhejiang University. Master in Artificial Intelligence.

1987-88 Assistant lecturer at Department of Computer Science, Zhejiang University.

1988 Attended Rutgers, The State University of New Jersey, New Brunswick.

1990 M.S. in Computer Science, Rutgers, The State University of New Jersey.

1991 Summer work on compiler development at Datacom Inc., Holmdel, New Jersey.

1993 Ph.D. in Computer Science, Rutgers, The State University of New Jersey.

1993- Assistant professor at Department of Computer Science, University of California at Santa Barbara.

Publications

1987 **1.** T. Yang, Z. He and R. Yu, Performance evaluation of the inference structure in expert systems, *Proc. of 10th International Joint Conference on Artificial Intelligence*, Italy, 1987, pp. 945-950.

1988 **2.** T. Yang, R. Yu and Z. He, The game tree functions and their optimizations, *Chinese Journal of Computer*, Vol. 20, No. 2 (1988), pp. 79-88.

1989 **3.** M. Rao, R. Cruz, T. Yang, and Y. Ying, Intelligent control for cell culture processes, *Proc. of 1989 American Control Conference*, Pittsburgh, PA, June 1989, pp. 2418-2423.

4. W. Wang, T. Yang, Z. Wu, Z. He, R. Yu, KMIX series: Towards an intelligent environment, *Proc. of 89 International Conf. on New Generation*

Computers, Beijing, International Academic Publishers, Apr. 1989, pp. 96-105.

1990 **5.** Z. Wu, T. Yang, F. Ni, Z. He and R. Yu, A C-oriented tool for building the second generation expert systems, *Future Generation Computer Systems*, Vol. 6 No. 1 (1990), pp. 77-83, North Holland Publisher.

6. A. Gerasoulis, S. Venugopal and T. Yang, Clustering task graphs for message passing architectures, *Proc. of 4th ACM International Conference on Supercomputing*, Amsterdam, June 1990, pp.447-456. Proceedings also published as *Computer Architecture News*, Vol. 18, No. 3, September, 1990.

1991 **7.** T. Yang and A. Gerasoulis, A fast static scheduling algorithm for DAGs on an unbounded number of processors, *Proc. of Supercomputing '91*, IEEE, Albuquerque, NM, Nov. 1991, pp. 633-642.

1992 **8.** T. Yang and A. Gerasoulis, A parallel programming tool for scheduling on distributed memory multiprocessors. *Proc. of 1992 Scalable High Performance Computing Conference*, IEEE, Williamsburg, VA., April, 1992, pp. 350-357.

9. T. Yang and A. Gerasoulis, PYRROS: Static scheduling and code generation for message passing multiprocessors. *Proc. of 6th ACM International Conference on Supercomputing*, Washington D.C., July, 1992, pp. 428-437.

10. A. Gerasoulis and T. Yang, Static scheduling of parallel programs for message passing architectures. *Lecture Notes in Computer Science, No. 634, Parallel Processing: CONPAR 92 – VAPP V*, L. Bouge, M. Cosnard, Y. Robert and D. Trystram (Eds.), Springer-Verlag, 1992, pp. 601-612.

11. A. Gerasoulis and T. Yang, A static macro-dataflow scheduling tool for scalable architectures. Invited paper in *Proc. of Summer School on Scheduling and its Applications*, Chateau de Bonas, France, Oct. 1992, pp. 382-417.

12. A. Gerasoulis and T. Yang, A comparison of clustering heuristics for

scheduling DAGs on multiprocessors. *Journal of Distributed and Parallel Computing*, Special issue on scheduling and load balancing, Vol 16, No. 4, Dec. 1992, pp. 276-291.

To appear **13.** A. Gerasoulis and T. Yang, Scheduling prorgam task graphs on MIMD architectures. To appear as a book chapter in *Algorithm Derivation and Program Transformation*, R. Paige, J. Reif, and R. Wachter (Eds), Kluwer Publisher.

14. T. Yang and A. Gerasoulis, List scheduling with and without communication. To appear in *Parallel Computing.*

15. A. Gerasoulis and T. Yang, On the granularity and clustering of directed acyclic task graphs. To appear in *IEEE Transactions on Parallel and Distributed Systems.*

Submitted for Publication

1993 **16.** T. Yang and A. Gerasoulis, DSC: Scheduling parallel tasks on an unbounded number of processors.

17. T. Yang and A. Gerasoulis, Code generation techniques for executing task graphs on message-passing architectures.