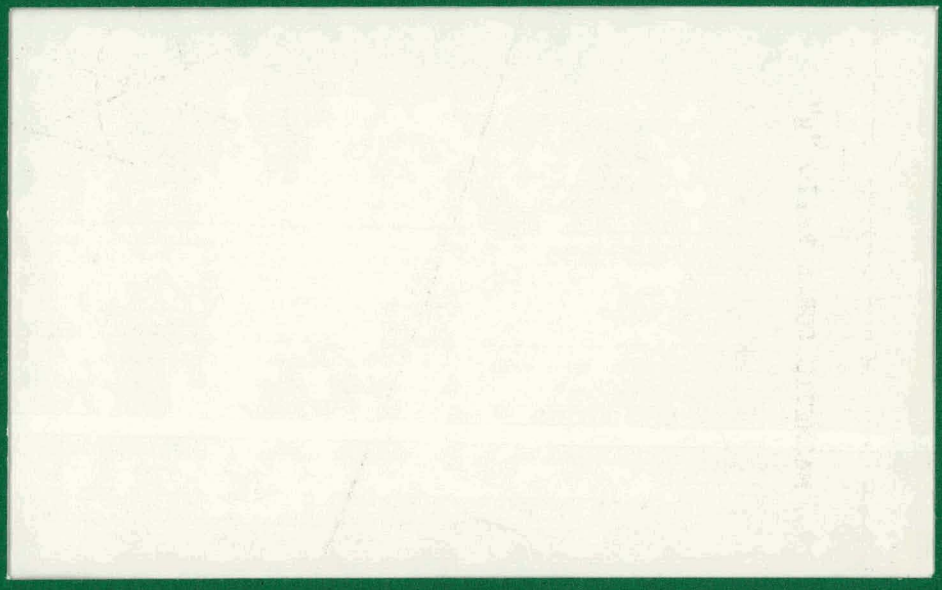
Center for Supercomputing Research & Development



Center for Supercomputing Research & Development
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign

DISCLAIMER

CSRD Rpt. No. 595

UILU-ENG-86-8006

DOE/ER/25001--85

DE88 003532

ON PROGRAM RESTRUCTURING, SCHEDULING, AND COMMUNICATION FOR PARALLEL PROCESSOR SYSTEMS

Constantine D. Polychronopoulos

August 1986

Center for Supercomputing Research and Development
University of Illinois
305 Talbot - 104 South Wright Street
Urbana, IL 61801-2932
Phone: (217) 333-6223

This work was supported in part by the National Science Foundation under Grants No. US NSF DCR84-10110 and US NSF DCR84-06916, the U. S. Department of Energy under Grant No. US DOE-DE-FG02-85ER25001, the IBM Donation and Alliant Computer Corporation, and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science, August 1986.

ON PROGRAM RESTRUCTURING, SCHEDULING, AND COMMUNICATION FOR PARALLEL PROCESSOR SYSTEMS

BY

CONSTANTINE D. POLYCHRONOPOULOS

B.Sc., National and Kapodistrian University of Athens, 1980
M.S., Vanderbilt University, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1986

Urbana, Illinois

Dedicated to my parents, Demetrios and Ekaterini,
to my wife Gerasimoula, and to my daughter Ekaterini.

ON PROGRAM RESTRUCTURING, SCHEDULING, AND COMMUNICATION FOR PARALLEL PROCESSOR SYSTEMS

Constantine D. Polychronopoulos, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1986
David J. Kuck, Advisor

This dissertation discusses several software and hardware aspects of program execution on large-scale, high-performance parallel processor systems. The issues covered are program restructuring, partitioning, scheduling and interprocessor communication, synchronization, and hardware design issues of specialized units. All this work was performed focusing on a single goal: to maximize program speedup, or equivalently, to minimize parallel execution time. Parafrase, a Fortran restructuring compiler was used to transform programs in a parallel form and conduct experiments. Two new program restructuring techniques are presented, *loop coalescing* and *subscript blocking*. Compile-time and run-time scheduling schemes are covered extensively. Depending on the program construct, these algorithms generate optimal or near-optimal schedules. For the case of arbitrarily nested hybrid loops, two optimal scheduling algorithms for dynamic and static scheduling are presented. Simulation results are given for a new dynamic scheduling algorithm. The performance of this algorithm is compared to that of self-scheduling. Techniques for program partitioning and minimization of interprocessor communication for idealized program models and for real Fortran programs are also discussed. The close relationship between scheduling, interprocessor communication, and synchronization becomes apparent at several points in this work. Finally, the impact of various types of overhead on program speedup and experimental results are presented.

ACKNOWLEDGEMENTS

First of all I would like to express my gratitude to my advisor Professor David J. Kuck. It is a privilege for having been one of his students. He constantly provided me with guidance, support, and encouragement throughout my graduate studies at Illinois. His ability to grasp complex but otherwise vague concepts and extract the important aspects, has greatly influenced my approach to research. I am indebted to him for all this. Professor Duncan Lawrie has always been a source of inspiration and I thank him for his invaluable help during my first years and up to the present. I am also grateful to David Padua for always being there to discuss technical and nontechnical issues; he often sacrificed precious time to talk with me. I have learned a great deal from David and I also thank him for being a good friend. I thank Professor Ahmed Sameh for his help during my first years. I would also like to thank my friends Ron Cytron, Tim McDaniel, and Alex Veidenbaum; without them Parafrase would still be a mystery to me. Utpal Banerjee deserves my special thanks for his help in sorting out "messy" ideas and for many stimulating discussions. I thank the members of my committee Professors Dave Liu and Dennis Mickunas for their advise. Many thanks are also due to my friends Anthony, Dan, Dave, Gyungho, Jayasimha, Harlan, Luddy, Perry, Sam, Steve, Todd, and William. Annetta, Barb, Becky, Dorothy, Gayanne, and Vivian were always willing to help me when bureaucracy was getting in my way. My parents Demetrios and Katerina, and my brothers Eleftherios, Yannis, and George kept me going with their constant love and support. Especially I would like to thank two very special people in my life, my wife Gerasimoula and my daughter Katerina. My wife for standing by me whenever I needed support of any kind. Together we went through times that only she could survive. My sweet daughter for arriving in this world at just the right time and bringing

me such great happiness. I deeply thank all of you.

> *...for it is not a thing that can be put into words. But from long communing together over the thing itself, and from the association of teacher and student, suddenly, like a flame leaping a gap, illumination kindles in the soul; and after that, it finds its own nourishment.*
>
> Plato (*Letters:* VII314c-341d.)

TABLE OF CONTENTS

CHAPTER

CHAPTER 1

INTRODUCTION

As technology approaches certain physical limitations, parallelism seems to be the most promising alternative for satisfying the ever-increasing demand for computational speed. The main driving force behind the development of parallel processor systems is the ability to exploit the parallelism in algorithms and programs, and solve problems whose computational complexity makes them impossible to tackle on conventional systems. Recently it has become clear that the shared memory parallel processor model will be one of the dominant architectures for the near future supercomputers. The flexibility, scalability, and high potential performance offered by parallel processor machines are simply necessary "ingredients" for any high performance system. The flexibility of these machines is indeed greater than that of single array processor computers [Kuck84], and they can execute more efficiently a larger spectrum of programs.

But there are divided opinions when the question comes to the number of processors needed for an efficient and cost-effective, yet very fast polyprocessor system. A number of pessimistic and optimistic reports have come out on this topic. One side uses Amdahl's law and intuition to argue against large systems [Mins70], [Amda67]. The other side cites simulations and real examples to support the belief that highly parallel systems with large numbers of processors are practical, and could be efficiently utilized to give substantial speedups [PoBa86], [Cytr84], [Krus84], [Kuck84], [Bane81], [FlHc80]. However, we have little experience in efficiently using a large number of processors. This inexperience in turn is reflected in the small number of processors used in modern commercially available supercomputers such as the CRAY X-MP, CRAY 2, and Alliant FX/8 systems.

Truly parallel languages, parallel algorithms, and ways of defining and exploiting program parallelism are still in their infancy. Only recently the appropriate attention has been focused on research for parallel algorithms, languages, and software. Several factors should be considered when designing high performance supercomputers [Kuck84]. Parallel algorithms, carefully designed parallel architectures and powerful programming environments including sophisticated restructuring compilers, all play equally important roles on program performance. In addition several crucial problems in scheduling, synchronization, and communication must be adequately solved in order to take full advantage of the inherent flexibility of parallel processor systems. The investment in traditional (serial) software however is so enormous, that it will be many years before parallel software dominates. It is then natural to ask: " How can we efficiently run existing software on parallel processor systems?" The answer to this question is well-known: by using powerful restructuring compilers. One such powerful restructurer is the Parafrase compiler developed over the last fifteen years at the University of Illinois ([KKLW80], [Wolf82]).

The work in this thesis involves several aspects of parallel processing. The primary challenge with parallel processor systems is to speed up the execution of a single program at a time, or maximize program speedup (as opposed to minimizing response time, or maximizing throughput). One of the most critical issues in parallel processing is the design of processor allocation and scheduling schemes that minimize execution time and interprocessor communication for a given program. A significant amount of theoretical work has been done on the subject of scheduling, but because of the complexity of these problems, only a few simple cases have been solved optimally in polynomial time. Moreover, almost none of these cases is of practical use. Another important issue is program partitioning. Given a parallel program, we need to partition it into a set of independent or communicating processes or tasks. Each process can then be allocated (scheduled on) one or more processors. Program partitioning affects and can be affected by

several factors. It is a multidimensional optimization problem where the variables to be optimized are not compatible.

Although loops are the largest potential source of program parallelism, the problem of using several processors for the fast execution of complex parallel loops had not been given enough attention until recently [PoKP86], [PaKL80], [Cytr84]. A key problem in designing and using large parallel processor systems is determining how to schedule independent processors to execute a parallel program as fast as possible. We know little about coordinating large numbers of processors to execute multiply nested parallel loops, and no significant work has been done thus far to adequately solve this problem.

1.1. Thesis Overview and Related Work

This thesis discusses and proposes solutions to some important problems that arise in parallel processing. Speedup models, scheduling, program restructuring, and program partitioning are the topics involved. More specifically, Chapter 2 considers three models of program execution and their associated speedup bounds. A generalization of the Doacross model [Cytr84] is also presented in Chapter 2. Chapter 3 discusses the issue of program partitioning and minimization of interprocessor communication. An idealized model for interprocessor communication is developed and then is applied to Fortran programs. Chapter 4 covers the topic of static processor allocation to multiply nested parallel loops. Optimal algorithms for simple and complex loops are presented. Chapter 5 presents a compiler transformation that is beneficial to both static and dynamic scheduling. This transformation can also be used in certain cases to improve memory management. The general problem of scheduling independent tasks on a parallel processor system and proposed solutions are covered in Chapter 6. Dynamic scheduling of complex loops and independent tasks is discussed in Chapter 7. A powerful dynamic scheduling algorithm for arbitrarily nested loops is presented. Simulation results for this algorithm and for self-

scheduling are also presented in Chapter 7. A new scheme for parallelizing loops with subscripted subscripts is presented in Chapter 8. Finally Chapter 9 gives the conclusion of this thesis.

Our work on scheduling differs from the previous work in several aspects. Instead of considering simplified abstract models [CoGJ78] [Coff76] [KaNa84], we focus on the aspects of scheduling of real programs. The central issue in this thesis is to develop static and dynamic scheduling schemes for arbitrarily nested Fortran loops. Little work has been reported on this topic so far [Cytr84], [PoKP86], [TaYe86], although it becomes an area of great theoretical and practical interest [Bohk85], [GGKM83]. This thesis presents optimal static and dynamic solutions for the general problem. The most significant work on scheduling parallel loops has so far been conducted at industrial laboratories. Microtasking [Rein86] for example is used to schedule parallel loops on the CRAY machines. However most of these efforts consider the simple case of singly nested parallel loops. No significant work has been reported for the case of multiply nested loops until recently [PoKP86] [TaYe86]. This is partially justified by the small number of processors used in real supercomputers; one level of parallelism would be adequate to utilize all processors.

A few dynamic scheduling schemes have been proposed in the past few years [KrWe85], [Mann84], [Bain78]. Most of these schemes are based on specialized scheduling hardware or operating system functions. In the case of hardware schemes the drawbacks are cost and generality. The majority of these schemes are designed for special purpose machines [Bain78], or for scheduling special types of tasks, e.g. atomic operations. On the other hand, the disadvantage of dynamic scheduling by the operating system is the high overhead involved. Especially when the granularity of tasks is small, the overhead involved with the invocation of the operating system is likely to outweigh the benefit of parallelism. It is more appropriate to implement dynamic scheduling without involving the operating system, by using low level primitives inside the pro-

gram. The most appropriate primitives are naturally various kinds of synchronization instructions. More recently a lot of attention has been focused on dynamic scheduling through the use of synchronization [TaYe86] [GGKM83]. These schemes however involve high run-time overhead and are not efficient for parallel loops with a complex nest pattern. An algorithm that is presented in Chapter 7 deals with such cases efficiently and involves minimal overhead.

Extensive work has been done on the problem of scheduling independent tasks on parallel processors [Sahn84], [Liu81], [CoGJ78], [Grah72], [CoGr72]. Most of the instances of this problem have been proved to be NP-Complete problems. Optimal algorithms have been discovered for special cases of the problem that restrict the tasks to be of unit-execution time and/or the number of processors to be 2. These theoretical results however are of little help to practical cases. Heuristic algorithms for approximate solutions [CoGJ78], [CoGr72], also use simplifying assumptions that make them difficult to use in practice. In Chapter 6 of this thesis we consider the general scheduling problem. By considering the nature of tasks that occur in Fortran programs we were able to design optimal algorithms and approximation heuristics that can be used efficiently in practice.

As mentioned earlier, a vast amount of existing software has been coded either in a serial language (e.g. Fortran, Pascal, Lisp, C), or for a serial machine. The need to run serial software on array or parallel machines without reprogramming gave rise to a new research field: *program restructuring*. During program restructuring, a compiler or preprocessor identifies the parts of a program that can take advantage of the architectural characteristics of a machine. Two equally important reasons for program restructuring are ease of programming and complexity. Coding a particular problem to take full advantage of the machine characteristics is a complex and tedious task. For non-trivial programs and on the average, a compiler can perform better than a skillful programmer. As is the case with traditional code optimization, restructuring can be automated.

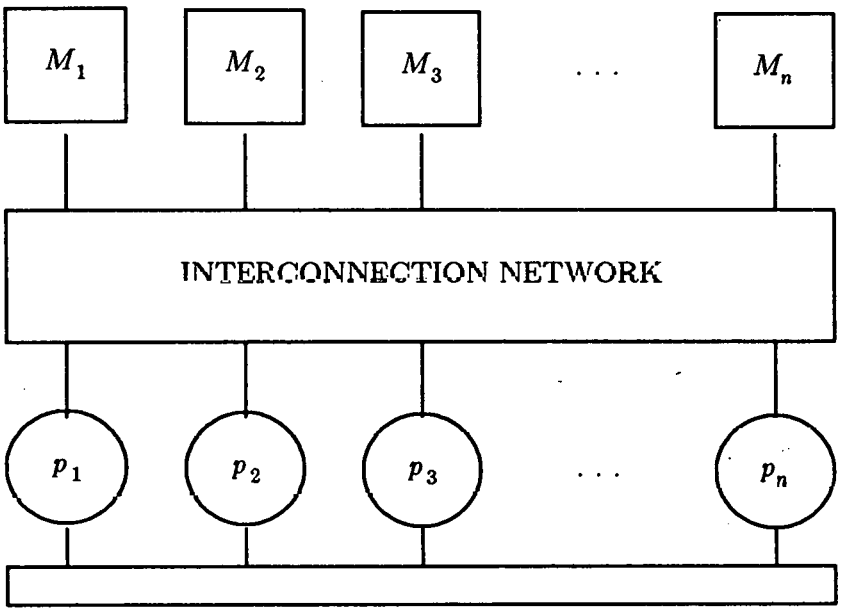


Figure 1.1. The shared memory parallel processor model.

A very significant amount of work has been performed on program restructuring primarily in the case of Fortran [KLPL81], [Kuck80], [Kenn80]. However more remains to be done in this area. Chapters 5 and 8 of this thesis present two restructuring schemes that can be used to extract more parallelism out of serial Fortran programs and obtain better schedules.

The next section introduces the basic assumptions, notation, and definitions that are used throughout this thesis. Notation and definitions that are relevant only to a particular section are given wherever appropriate. A brief overview of the Parafrase compiler that was used for experiments is also given. Most of the experimental work discussed in this thesis was performed and designed in the context of Parafrase (Figure 1.2).

1.2. Basic Concepts and Definitions

The architecture model used throughout this thesis is a shared memory parallel processor system as shown in Figure 1.1. The machine consists of p processors (numbered 1,2,...,p) that are connected to a shared memory M through a multistage interconnection network N. The memory can be interleaved and each processor can access any memory module, or can communicate with any other processor through the memory. Each processor has its own private memory that can be organized as a cache, register file or RAM. Each processor is a stand-alone unit. It has its own control unit and can operate independently and asynchronously from the other processors. Our machine model therefore is multiple-instruction, multiple-data or MIMD. We also assume that each of the processors is a vector or array processor and thus it can operate in single-instruction, multiple-data or SIMD mode. It is apparent that this taxonomy [Flyn80] cannot uniquely characterize our machine model. Another taxonomy proposed in [Kuck78] is used later to describe the machine model of Figure 1.1.

In this thesis we consider parallel Fortran programs. By parallel, we mean programs that have been written using language extensions or programs that have been restructured by an optimizing compiler. For our purposes we use output generated by the Parafrase restructurer [Kuck80], [Wolf82]. Parafrase is a restructuring compiler which receives as input Fortran programs and applies to them a series of machine independent and machine dependent transformations. The structure of Parafrase appears in Figure 1.2. The first part of the compiler consists of a set of machine independent transformations (passes). The second part consists of a series of machine dependent optimizations that can be applied on a given program. Depending on the architecture of the machine we intend to use, we choose the appropriate set of passes to perform transformations targeted to the underlying architecture. Currently Parafrase can be used to transform programs for execution on four types of machines: *Single Execution Scalar* or SES

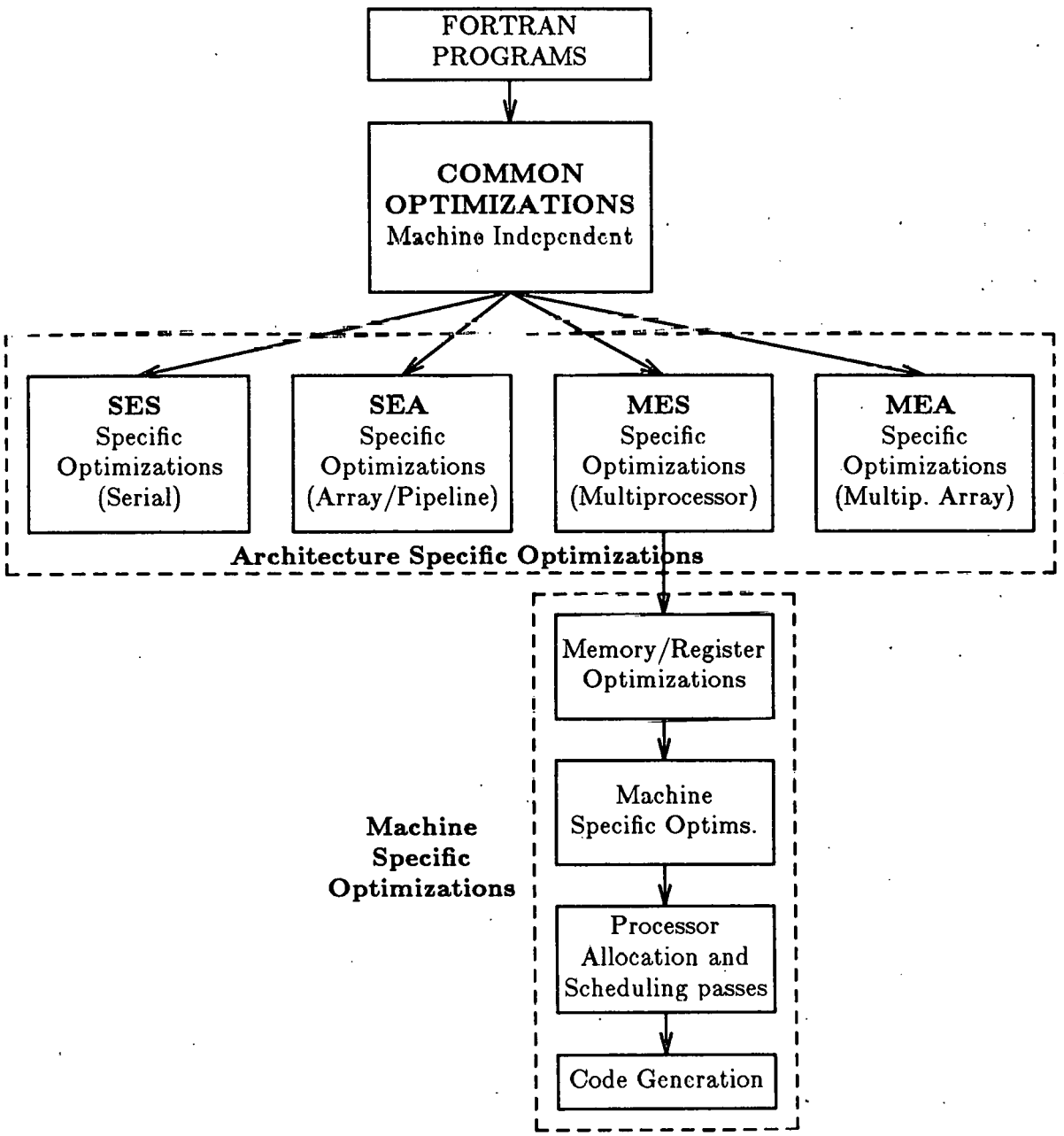


Figure 1.2. The structure of PARAFRASE

(uniprocessor), *Single Execution Array* or SEA (array/pipeline), *Multiple Execution Scalar* or MES (multiprocessor), and *Multiple Execution Array* or MEA (multiprocessor with vector

processors) architectures [Kuck78]. SES is a uniprocessor machine and optimizations for serial architectures include the traditional code optimizations used in most compilers [AhUl77]. SEA architectures include all single instruction multiple data models such as vector or pipeline and array machines. The MES model includes parallel processor systems with serial processors, or more commonly referred to as MIMD. In other words MES systems are composed of a set of independent SES machines (that may operate out of a shared memory). In case of a parallel processor system where each processor has an SEA or SIMD organization, the corresponding machine is called MEA. The machine models used for this thesis are MES and MEA. The front-end passes or transformations used in Parafrase are applicable to all machine organizations defined above. In addition Parafrase has a set of back-end transformations for each different architecture.

The most important aspect of the restructurer is its ability to perform sophisticated dependence analysis and build the data dependence graph (DDG) of a Fortran program. The DDG is an internal representation of the program that is used by most subsequent passes to carry out a variety of transformations and optimizations without violating the semantics of the source program. The DDG is a directed graph, where nodes correspond to program statements and arcs represent dependences. Dependences in turn enforce a partial order of execution on the program statements. For most of our work we make an implicit assumption that Parafrase can supply us with any *compact data dependence graph* or CDDG. A CDDG is a directed graph that can be built from the DDG by condensing several nodes into a single *composite node*. A set of consecutive program statements for example can define a composite node. The arcs in DDG's represent collections of dependences. Clearly several different CDDG's can be constructed from the same DDG. As we see later, Parafrase builds CDDG's that are used by certain transformations.

There are four different types of arcs or *dependences* in a DDG. Let s_i and s_j be two program statements, and suppose that during serial execution s_i is executed before s_j. If s_i and s_j are involved in a *use-definition* chain [AhUl77], where a variable defined in s_i is used in s_j (and is not reassigned in between), then we say that s_j is *data* or *flow dependent* on s_i. If a variable assigned in s_j is used in s_i, then there is an *anti-dependence* from s_i to s_j. If the same variable is assigned in both s_i and s_j (and not in between), then s_j is *output dependent* on s_i. Finally the fourth type of dependences are *control dependences* that originate from decision (conditional) nodes and point to the statements of the target code. More about dependence analysis can be found in [Bane79], [Bane76]. We often use the term "data dependence" to refer to any type of dependence.

In a restructured Fortran program we observe several types of parallelism and all of them can be potentially utilized by an MES machine. We can roughly classify the different types of parallelism into two categories: *Fine grain parallelism* and *coarse grain parallelism*. Fine grain parallelism includes the parallel execution of different statements of the program on different processors, or even different operations of the same statement on different processors or functional units. Coarse grain parallelism arises from the parallel execution of independent disjoint modules of the program, or from parallel loops.

Through a series of transformations, Parafrase is able to restructure Fortran loops into a parallel form. There are three major types of loops in a typical restructured program. *Doserial* (or DOSERIAL) are loops that must execute serially on any machine due to dependence cycles. An important transformation in the restructurer is the *do-to-doall* pass, that recognizes and marks *Doall* (or DOALL) loops. In a Doall loop cross-iteration dependences do not exist and thus all iterations can execute in parallel and in any order. A restricted case of a Doall is the *Forall* loop. In a Forall loop cross-iteration dependences of constant distance (usually greater

than one) may exist, but strip mining can be used to execute such loops as Doalls. Another pass in Parafrase is the *do-to-doacross* that recognizes and marks *Doacross* (or DOACR) loops. A DOACR is a loop that contains a dependence cycle in its loop body. If the cycle involves all statements in the loop a DOACR is then equivalent to a DOSERIAL loop. Otherwise partial overlapping of successive iterations may be possible during execution. A DOALL loop can also be thought of as a special case of a DOACR without dependence cycle in its body.

In addition to restructuring Fortran programs Parafrase supplies the user with program statistics that include speedup of execution for different numbers of processors, parallel and serial execution times, and efficiency and utilization measures. For a Fortran program, T_1 denotes its serial execution time. T_p denotes the parallel execution time of a program on a p processor machine. For a given p, the *program speedup* S_p is then defined as

$$S_p = \frac{T_1}{T_p}$$

The *efficiency* E_p for a given program and a given p is defined as the ratio,

$$E_p = \frac{S_p}{p}$$

and $0 \leq E_p \leq 1$. Often in this thesis the terms *parallel execution time* and *schedule length* are used interchangeably.

The overlapped execution of disjoint modules of a parallel program is referred to as *spreading*. If spreading is performed for *fine grain* parallelism it is called *low-level spreading*. If high level or coarse grain parallelism (e.g. disjoint loops) is used it is called *high-level spreading* [Veid85]. A *block of assignment statements* (or BAS) is a program module consisting exclusively of consecutive assignment statements. A BAS is also referred to as a *basic block* [AhUl76]. Basic blocks have a single entry and a single exit statement which are the first and last statements in

the BAS. A *program task graph* is any compact DDG. Nodes in a program task graph correspond to program modules and are called *tasks*. Arcs represent dependences and are labeled with *weights* reflecting the amount of data that need to be transmitted between tasks. The arcs of a CDDG define a partial order on its nodes. The *predecessors* of a task are the nodes pointing to that task. The tasks pointed to by the arcs originating from a given node are *successors* of that node. Tasks may be *serial* or *parallel* depending on whether they can execute on one or more processors. A parallel task may *fork* or *spawn* several *processes* with each process executing on a different processor. Serial tasks are composed of a single process. A task is said to be *active* if it can spawn more processes, or if some of its processes have not completed execution. A task is *ready* when it does not have predecessor tasks, and it is *complete* when all of its processes have completed execution.

A *processor allocation* is the assignment of a number of processors to each task of the program task graph. A *schedule* is the assignment of tasks to processors under time constraints. Note that processor allocation specifies the number of processors assigned to each task but not the actual binding of tasks to physical processors. Scheduling on the other hand binds a specific task to one or more physical processors at a specific time interval. Processor allocation takes into account the timing constraints implicitly. A schedule may be *static* or *dynamic*. A static schedule specifies the assignment of tasks to processors deterministically before execution. Dynamic scheduling performs the binding dynamically at run-time in a nondeterministic approach. A variation of dynamic scheduling is *self-scheduling*. During self-scheduling idle processors fetch their next process from a shared pool of ready tasks.

A *schedule-length* is the time it takes to execute a given program (graph) under a specific scheduling scheme. If a program starts executing at time 1, the execution time is determined by the moment the last processor working on that program finishes. For the same program graph

different scheduling schemes have (in general) different schedule lengths. *Finish time* and *completion time* are synonyms to schedule length.

A program variable is *shared* if it is used in more than one process (and is not a read-only variable). A *barrier* is a shared variable that assumes integer values in a fixed range $[N, M]$, where N, M are integers and $N < M$. A *barrier synchronization* is a synchronization primitive that tests and performs increment/decrement operations on a barrier.

CHAPTER 2

SPEEDUP BOUNDS FOR PARALLEL PROGRAMS

In this chapter we start with a parallel program that is the result of restructuring a serial program for execution on a parallel processor machine. We discuss different types of parallelism that can be observed in such a program, the kinds of overhead involved during parallel program execution, and the effect of task size and scheduling overhead on speedup. We then address the problem of allocating available processors to different parts of the program and estimating the possible speedup. Program task graphs have been chosen to provide concrete representations of parallel programs. These are directed graphs where a node represents a DOACR loop and arcs represent precedence constraints.

We have restructured LINPACK using Parafrase and computed the fraction of parallel code for all subroutines. Our results contradict the original Amdahl conjecture that most programs have at least 10% serial code, and hence can achieve a maximum speedup of 10. The experiments strongly support the view that there is enough inherent parallelism in real programs so that large numbers of processors can be efficiently utilized.

2.1. Basic Concepts

A basic sequential machine is a single CPU computer that can carry out operations serially, taking one unit of time for each. A p-unit multiple execution scalar (MES) machine is composed of p identical basic sequential machines, and each processor is driven by its own control unit. Because of its flexibility, we will consider only the MES machine. It is variously referred to as a multiprocessor, a parallel machine with p-processors, or simply a p-processor machine.

An assignment statement is a statement of the form $x = E$, where x is a variable and E an expression. A *do across* or DOACR loop [Cytr84] with *delay* d has the form

$$\begin{array}{l} L : \text{DOACR } I = 1, N \\ \quad\quad B \\ \text{END} \end{array}$$

where d, N are integer constants, I an integer variable with the range $\{1, 2,..., N\}$, B a sequence of assignment statements and DOACR loops, and it is understood that the iterations of L can be partially overlapped as long as there is a delay of at least d units of time from the start of iteration i to the start of iteration $i + 1$, $(i = 1, 2,..., N - 1)$. For L, the index variable is I, the number of iterations is N, and the loop-body is B.

Consider now the two extreme cases of overlapping. If $d = 0$, there is complete overlapping, i.e. all the iterations of L can be executed simultaneously. In this case the DOACR loop is called a DOALL loop. If $d = b$, where b is the execution time (assumed to be independent of I) of the loop-body B, then there is no overlapping, i.e. the iterations of L must be executed serially, one after another. In this case the DOACR loop is a standard serial loop, and we write DOSERIAL for DOACR. A BAS or a block of assignment statements is a special kind of serial loop, namely a loop with a single iteration. (A BAS may also be regarded as a special case of a do all loop.)

A program is a sequence of steps where each step consists of one or more operations that can be executed simultaneously. A program is *serial* if each step has exactly one operation; otherwise it is *parallel.* Two programs are semantically *equivalent* if they always generate the same output on the same input. Parallel programs are conveniently represented in terms of do across loops (Section 2.5).

Let $PROG_1$, $PROG_2$ be two equivalent programs and let their execution times on a p-processor machine be $T_p(PROG_1)$ and $T_p(PROG_2)$ respectively. Then the *speedup* obtained

(on this machine) by executing $PROG_2$ instead of $PROG_1$ is denoted by $S_p(PROG_1, PROG_2)$ and is defined by

$$S_p(PROG_1, PROG_2) = \frac{T_p(PROG_1)}{T_p(PROG_2)}.$$

An immediate consequence of this definition is the following lemma.

Lemma 2.1. If $PROG_1$, $PROG_2$, ..., $PROG_n$ is a sequence of programs any two of which are equivalent, then

$$S_p(PROG_1, PROG_n) = \prod_{i=1}^{n-1} S_p(PROG_i, PROG_{i+1}).$$

We usually write S_p for the speedup when the two programs involved are understood. Of special interest to us is the case where $PROG_1$ is a serial program and $PROG_2$ is an equivalent parallel program obtained by restructuring $PROG_1$. In this chapter we assume that the execution time of a program is determined solely by the time taken to perform its operations, and that the total number of operations in a program is never affected by any restructuring. These assumptions are not very far from the truth; they help to keep the formulas simple, and yet let us derive important conclusions. If T_1 is the number of operations in the serial program $PROG_1$, then T_1 is also the execution time of $PROG_1$ on the basic sequential machine, or on any parallel machine (i.e., $T_1 = T_p(PROG_1)$). The equivalent parallel program $PROG_2$ also has T_1 operations, but now these operations are arranged in fewer than T_1 steps. We call T_1 the *serial execution time* of $PROG_2$ and it can be obtained simply by counting the operations in $PROG_2$. The execution time $T_p \equiv T_p(PROG_2)$ of $PROG_2$ on the p-processor machine will depend on the structure of the program, the magnitude of p, and the way the p processors are allocated to different parts of $PROG_2$. To distinguish it from T_1, T_p is referred to as the *parallel execution time* of $PROG_2$. The speedup of a program is then the ratio of its serial execution time to its

parallel execution time.

For a given program, we have the *unlimited* processor case when p is large enough so that we can always allocate as many processors as we please. Otherwise, we have the *limited* processor case. These two cases will be often discussed separately.

There are several factors affecting the speedup of a given program. For example, different compiler implementations or different compiler algorithms used on the same problem may result in different speedups. Given a particular parallel machine $M_i \in M$, where M is the universal set of machine architectures, and a set A of equivalent algorithms (all of which receive the same input and produce the same output), we can define a mapping:

$$E_{M_i}: \quad A \rightarrow R_0^+$$

where R_0^+ is the set of the nonnegative real numbers and $E_{M_i}(A_j) = T_j$ is the execution time of algorithm A_j on machine M_i. Let $A_0 \in A$ be the algorithm for which $E_{M_i}(A_0) = \min_j\{E_{M_i}(A_j)\}$ and A_c be the algorithm we currently have available. Then the speedup we can achieve by selecting A_0, (the most appropriate algorithm for the specific architecture) would be:

$$S_c = \frac{E_{M_i}(A_c)}{E_{M_i}(A_0)}$$

The selection of the fastest algorithm is the user's responsibility and it seems unlikely that this process will be automated at least in the foreseeable future.

2.2. Restructuring, Program Partitioning and Critical Task Size

In our program model we assume that parallelism is explicitly specified in the form of tasks (disjoint code segments) which are parallel loops (DOALL or DOACR). This can be done for any Fortran program written in a serial form by employing restructuring compilers. In our case

Parafrase was used to transform programs into parallel form and compute some experimental values presented in the following section. Each branch of an IF or GOTO statement is assigned a *branching probability* by the user, or automatically by Parafrase [Kuck84]. We can therefore view any program as a sequence of assignment statements, where each statement has an accumulated *weight* associated with it. All loops in a program are automatically *normalized,* i.e., loop indeces assume values in [1, N] for some integer N. As in the case of branching statements, unknown loop upper bounds are either defined by the user, or automatically by the compiler (using a default value). During parallel execution of the restructured program, data and control dependencies must be observed to assure that program semantics is preserved. For this reason, the data dependence graph of the program is used by most transformations as a guide.

If we consider a block of assignment statements as a loop with a single iteration, a restructured program can be viewed as a series of outermost DOACR loops with each such loop being arbitrarily complex. This defines a "natural" partition of a restructured program into a series of code segments or tasks. Dependencies may exist between any pair of segments in the program. We can thus define the program task graph as a directed graph $G(V, E)$, where the nodes in V are the outermost loops L_i in the program, and there is an arc from a node L_i to a node L_j if and only if loop L_j depends on loop L_i. Since backward dependencies are not allowed, $G(V, E)$ is acyclic.

In a restructured program we may observe two types of parallelism: *horizontal* and *vertical.* Horizontal parallelism results by executing a DOACR loop on two or more processors, or equivalently, by simultaneously executing different iterations of the same loop. Vertical parallelism in turn, is the result of the simultaneous execution of two or more different loops (tasks). Two or more loops can execute simultaneously only if there exists no control or data dependencies between any two of the loops. In the general case the program task graph exposes both

types of parallelism. When we execute such a task graph on an MES machine, we must decide how to allocate the available processors to the program tasks so that program speedup is maximized.

A serious problem arises when while executing a restructured program on an MES machine, we attempt to minimize the overheads of communication, synchronization and scheduling. This is a non-trivial optimization problem, and attempts to minimize such overheads usually results in reducing the degree of program parallelism. Most instances of this optimization problem have been proven to be NP-Complete [GaJo78]. A heuristic algorithm would attempt to minimize the communication cost by merging nodes of the graph together to avoid the overhead involved in communicating data from one processor to another. This however often reduces the degree of available vertical parallelism (Chapter 3).

As an example of node merging, consider two loops L_1 and L_2 in our restructured program model, with data dependencies going from L_1 to L_2. The dependencies restrict the two loops to execute in this order since data computed in L_1 are used by L_2. In this case only horizontal parallelism inside each loop can be exploited. If we do not coordinate the processors chosen for the execution of L_1 and L_2, then data computed inside L_1 will have to be stored in a shared memory upon completion of L_1, and then fetched from that memory to the processors executing L_2. If on the other hand we consider the two loops as a single task, then we can bind iterations of L_1 and corresponding iterations of L_2 to specific processors. In this manner data computed by a particular iteration of L_1 and used by the corresponding iteration of L_2 need only be stored in fast registers of the processor, thus avoiding the overhead of redundant store and fetch operations. For relatively small loops the savings by such "task merging" can be very significant.

Task merging can also be used to decrease scheduling overhead that is involved when we distribute different program nodes across different processors. This scheduling overhead is in

addition to the synchronization overhead and may become disastrous especially for very small tasks. For the CRAY X-MP for example, the overhead involved with scheduling two parallel tasks can be several *msecs* [Cray85]. This overhead imposes a minimum size on parallel tasks, below which the speedup becomes rather a slowdown (i.e., $S_p < 1$). We call this the *critical task size.*

If during the execution of a program we schedule a set of parallel tasks, the parallel execution time is augmented by O_T, where O_T is the scheduling overhead. The maximum expected speedup therefore is given by

$$S_p = \frac{T_1}{T_1/p + O_T}.$$

In order to have a speedup of at least 1, we must have $T_1 \geq T_1 / p + O_T$, i.e., $T_1 \geq p*O_T / p-1$ which gives the critical task size as a function of the overhead and the number of processors. More generally, the minimum program size $T_{\min}$ required to obtain a given speedup S^* on p processors should satisfy:

$$\frac{T_{\min}}{T_{\min} / p + O_T} \geq S^*, \quad \text{or} \quad T_{\min} \geq \frac{p*O_T*S^*}{p - S^*}.$$

Program partitioning for minimizing data communication and scheduling overheads is a complicated optimization problem and it is the subject of the next chapter.

2.3. General Bounds on Speedup

In this section we consider an arbitrary parallel program, and think of it simply as a sequence of steps where each step consists of a set of operations that can execute in parallel. The total number of operations (and hence the serial execution time) is denoted by T_1. Let p_0 denote the maximum number of operations in any step.

Suppose first we are using a p-processor system with $p \geq p_0$. (This is the unlimited processor case). Let $\phi_i T_1$ denote the number of operations that belong to steps containing exactly i operations, ($i = 1, 2,..., p_0$). Then ϕ_i is the fraction of the program that can utilize exactly i processors, and we have $\sum_{i=1}^{p_0} \phi_i = 1$. We call $f = \sum_{i=2}^{p_0} \phi_i$ the *parallel part* of the program or the *fraction of parallel code,* and $1 - f = \phi_1$ the *serial part* of the program or the *fraction of serial code.* (At least $p - p_0$ processors will always remain unused.)

Consider now a limited processor situation with a p-processor machine where $p < p_0$. The steps with more than p operations have to be folded over and replaced with a larger number of steps with p operations. (For simplicity, we are assuming that each new step has exactly p operations, although one of them may actually have fewer than p). Let $f_i \equiv f_i(p)$ denote the fraction of the modified program that can utilize exactly i processors, ($i = 1, 2,..., p$). Then we have

$$f_i = \phi_i \quad (i = 1, 2,..., p - 1), \quad \text{and} \quad f_p = \sum_{i=p}^{p_0} \phi_i.$$

As long as $p \geq 2$, the parallel part f is given by $\sum_{i=2}^{p} f_i$ and the serial part $1 - f$ by f_1.

An arbitrary p-processor machine is assumed in the following. The first two results are well-known [Bane81], [Lee77].

Theorem 2.1. $$\frac{1}{S_p} = \sum_{i=1}^{p} \frac{f_i}{i}.$$

Proof: When executing on a p-processor machine, the fraction of the program that uses exactly i processors is $f_i T_1$, ($i = 1, 2,..., p$). Hence, the number of steps where i processors are active is $\frac{f_i T_1}{i}$. The total number of steps is then given by

$$T_p = \sum_{i=1}^{p} \frac{f_i T_1}{i} = T_1 \sum_{i=1}^{p} \frac{f_i}{i}.$$

Since T_1 is the serial and T_p the parallel execution time of the program, we get

$$\frac{1}{S_p} = \frac{T_p}{T_1} = \sum_{i=1}^{p} \frac{f_i}{i}. \quad \blacksquare$$

Corollary 2.1. $1 \leq S_p \leq p$.

Proof: We have $f_i \geq \frac{f_i}{i} \geq \frac{f_i}{p}$ $(i = 1, 2, \ldots, p)$. Hence

$$\sum_{i=1}^{p} f_i \geq \sum_{i=1}^{p} \frac{f_i}{i} \geq \sum_{i=1}^{p} \frac{f_i}{p}$$

$$\text{or,} \quad 1 \geq \sum_{i=1}^{p} \frac{f_i}{i} \geq \frac{1}{p}$$

$$\text{so that} \quad 1 \geq \frac{1}{S_p} \geq \frac{1}{p},$$

i.e. $1 \leq S_p \leq p$. $\blacksquare$

Corollary 2.2. $S_p \leq 1 / f_1$.

Corollary 2.3. The speedup S_p, the number of processors p and the fraction of parallel code f satisfy (for $p > 1$)

$$S_p \leq \frac{p}{f + (1 - f)p}, \tag{2.1}$$

$$f \geq \frac{S_p - 1}{S_p} \frac{p}{p - 1}, \tag{2.2}$$

$$\text{and} \quad p \geq \frac{fS_p}{1 - (1 - f)S_p}. \tag{2.3}$$

Proof: These three inequalities are equivalent; from any one the other two can be derived easily.

Note that

$$\sum_{i=1}^{p} \frac{f_i}{i} = f_1 + \sum_{i=2}^{p} \frac{f_i}{i} \geq f_1 + \sum_{i=2}^{p} \frac{f_i}{p} = 1 - f + \frac{f}{p},$$

since $f = \sum_{i=2}^{p} f_i$ and $f_1 = 1 - f$. Then by Theorem 2.1, $\frac{1}{S_p} \geq 1 - f + \frac{f}{p}$, so that

$$S_p \leq \frac{p}{f + (1 - f)p}. \blacksquare$$

Now, assume we have a program that can use a maximum number of p_0 processors. If the fraction ϕ_1 of serial code in it is very small, we can choose p (> 1) processors such that $f_p = \sum_{i=p}^{p_0} \phi_i \approx 1$. Then, since

$$\frac{1}{S_p} = \sum_{i=1}^{p-1} \frac{f_i}{i} + \frac{f_p}{p} \quad \text{we get}$$

$$\left| \frac{1}{S_p} - \frac{f_p}{p} \right| = \left| \sum_{i=1}^{p-1} \frac{f_i}{i} \right| \leq \sum_{i=1}^{p-1} f_i = 1 - f_p \approx 0,$$

or equivalently, $S_p \approx p / f_p \approx p$. Thus, if for some $p > 1$, $\sum_{i=p}^{p_0} \phi_i \approx 1$, then the program runs very efficiently on a p-processor system giving an almost linear speedup. In this case, given the coefficients ϕ_i for the particular program, we can always determine the maximum number of processors that would get a linear speedup.

Because of Corollary 2.2, Amdahl and some other researchers thereafter questioned the usefulness of very large MES systems, since, according to their argument, the majority of programs have an average of more than 10% serial code and therefore their speedup on any MES machine is bounded above by 10.

We conducted some experiments to measure the fraction of parallel code f in LINPACK, a widely used numerical package for solving systems of linear equations. Knowing the serial

Sub. Name	f	Sub. Name	f	Sub. Name	f
SPOFA	0.9997	SSIFA	0.9862	SPBFA	0.9257
SQRDC	0.9988	SPODI2	0.9853	SGBFA	0.9189
SPBDI1	0.9975	SGESL1	0.9807	SGBSL2	0.9164
SGBDI1	0.9974	SSISL	0.9806	SGBCO	0.8561
SGEDI2	0.9961	STRSL0	0.9773	SPBCO	0.8314
SQRDC1	0.9961	STRSL1	0.9773	SSIDI3	0.7353
SSIDI2	0.9961	SPOSL	0.9767	SGBSL1	0.6545
SSVDC1	0.9954	SGESL2	0.9762	STRCO	0.6113
SPODI1	0.9950	STRSL2	0.9753	SPBSL	0.5659
SSICO	0.9905	STRSL3	0.9753	SGTSL	0.5295
SQRSL1	0.9900	SPOCO	0.9751	SPPDI	0.5064
SQRSL2	0.9900	SPPCO	0.9746	SSPSL	0.4010
SQRSL4	0.9900	SSIDI1	0.9746	SPTSL	0.3799
SQRSL5	0.9900	SGEDI1	0.9745	SSPDI	0.3615
SSPCO	0.9896	SGECO	0.9664	SSPFA	0.1348
SQRSL3	0.9868	SPPSL	0.9629		
SGEFA	0.9862	SPPFA	0.9350		

Table 2.1. Values of f for LINPACK subroutines.

execution time T_1, the parallel execution time T_p and the number of processors p that were used during the execution of a subroutine, we can easily compute a lower bound for f from (2.2). All the above parameters are supplied by Parafrase. On the other hand, if the value of f for a particular subroutine is known and we want to achieve a specific speedup S_p for this subroutine, then (2.3) gives us a lower bound on the number of processors that we must use.

The sorted lower bounds of f are shown in Table 2.1. The measurements were done on LINPACK subroutines after they had been restructured by Parafrase. From Table 2.1 we observe that the majority of subroutines have a very high fraction of parallel code. For the first 37 subroutines (out of 49), the average fraction of parallel code was $\bar{f} \geq 0.9784$. Almost 76% of the subroutines have $f > 0.9$ and only 18% have $f < 0.8$.

Considering that LINPACK is a typical numerical package not very amenable to restructuring, the results of Table 2.1 are very encouraging. EISPACK for example (another numerical package), should be expected to have a much higher value of $\bar{f}$ than LINPACK [Kuck84]. Since several numerical packages are more amenable to restructuring than LINPACK, we should be more optimistic when designing large multiprocessor systems. The claim for the non-effectiveness of systems with large numbers of processors is mostly based on programs that exhibit an $f < 0.9$. As mentioned in Section 2.2, the real performance threat for large MES systems lies in scheduling and interprocessor communication overheads.

Secondly, we should consider all possible operating modes of a multiprocessor. There is no question that there exist numerical programs that could fully exploit hundreds or thousands of processors. For programs that utilize only a few processors, MES systems can be operated in a multiprogramming mode to keep system utilization high. The question then breaks down to whether we can have sites with enough users (workload) to keep system utilization at acceptable levels. The answer to this question is rather obvious.

2.4. Speedup and Processor Allocation for Task Graphs

We consider here an arbitrary parallel program represented by a task graph $G \equiv G(V, E)$. Recall from Section 2.2 that this graph is defined on a restructured program with nodes representing outermost DOACR loops, and arcs representing data and control dependencies among loops. Let there be n nodes in V: v_1, v_2, ..., v_n. These nodes can be partitioned into disjoint layers V_1, V_2, . . . , V_k, such that (1) all nodes in a given layer can execute in parallel, and (2) the nodes in a layer V_{i+1} can start executing as soon as all the nodes in layer V_i have finished, ($i = 1, 2, ..., k - 1$). To construct this layered graph of G, we use a modified Breadth First Search scheme for labeling the nodes of the graph. Initially, the first node of the

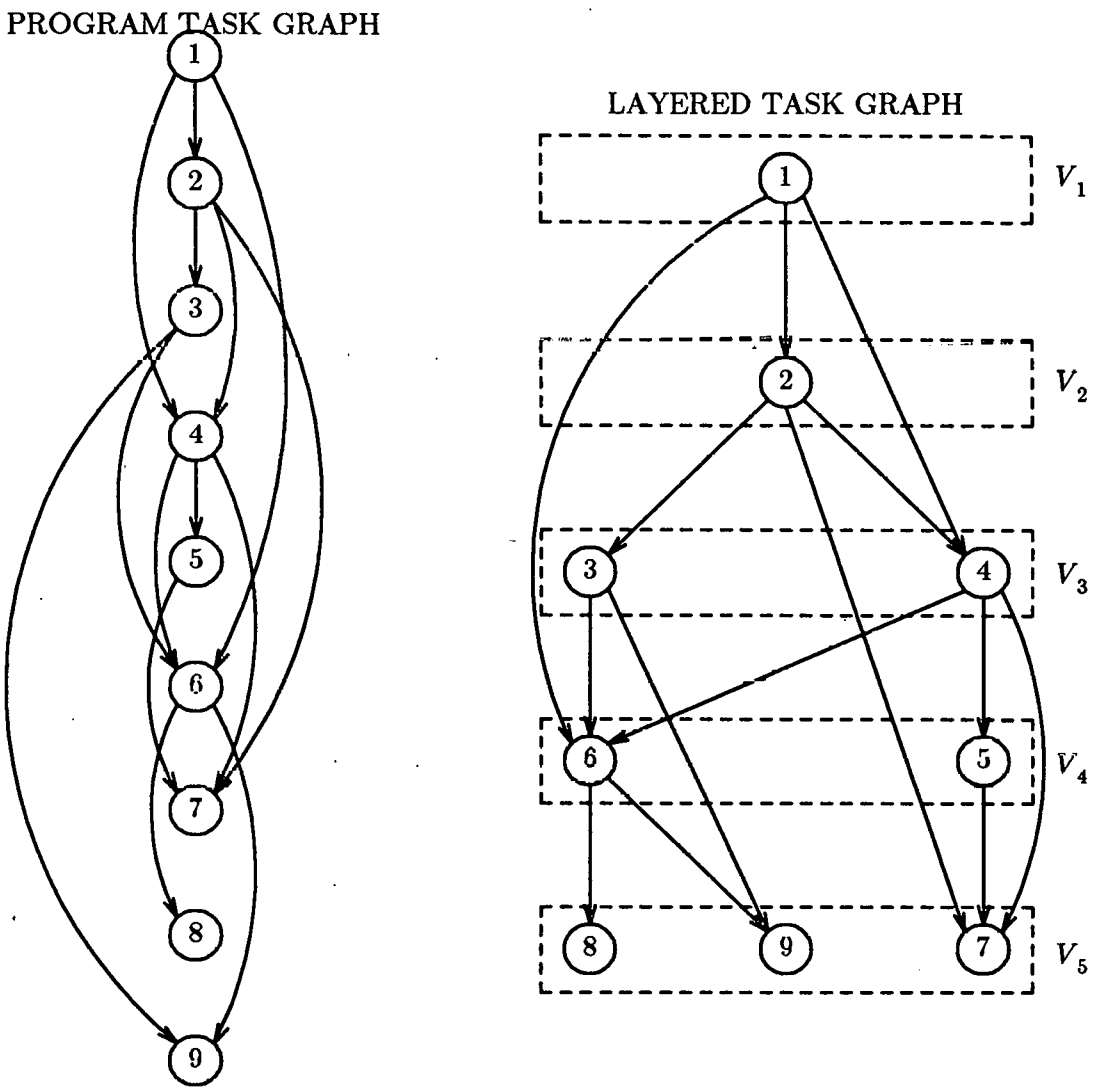


Figure 2.1. A example of a program task graph and its corresponding layered graph.

graph (corresponding to the lexically first loop of the program), is labeled 1 and queued in a FIFO queue Q. At each following step, v_j, the node at the front of Q is removed, and if i is its label, all nodes adjacent to v_j are labeled $i + 1$, and are queued in Q. Note that a node may

be relabeled several times but its final label is the largest assigned to it. When Q becomes empty, the labeling process terminates and we get the layered graph by grouping all nodes with label i into layer V_i. An example of a program task graph and its corresponding layered task graph is shown in Figure 2.1. We consider below three execution models of a layered task graph on a p-processor MES machine. The most general and the two extreme cases are discussed.

As usual T_1 denotes the total number of operations in the whole program. For a node v_j, let $g_j T_1$ denote the number of operations in the node and T_{pj} its parallel execution time, $(j = 1, 2,..., n)$. Then $\sum_{j=1}^{n} g_j = 1$. The *absolute speedup* S_{pj}^A of v_j is the speedup obtained by considering the node separately as a program and is given by $S_{pj}^A = g_j T_1 / T_{pj}$.

Case 1. (Horizontal parallelism). Let $k = n$ and each layer V_i consist of a single node v_i, $(i = 1, 2,..., n)$. To get the maximum speedup for the whole program on p processors, we need to get the maximum speedup for each node. Detailed formulas are given below.

The *relative speedup* S_{pi}^R of v_i is the speedup of the whole program when only v_i is executed in parallel and all other nodes are executed serially. Thus

$$S_{pi}^R = \frac{T_1}{T_1 - g_i T_1 + T_{pi}}.$$

Lemma 2.2. The absolute and relative speedups of a node v_i are connected by the equation

$$\frac{g_i}{S_{pi}^A} = \frac{1}{S_{pi}^R} - 1 + g_i \qquad (i = 1, 2,..., n).$$

Proof: It follows directly from the above definitions. ■

Theorem 2.2. The speedup S_p of the whole program (when all nodes are executed in parallel) is related to the absolute and relative speedups of the individual nodes by the following equations:

$$\frac{1}{S_p} = \sum_{i=1}^{n} \frac{g_i}{S_{pi}^A} = \sum_{i=1}^{n} \frac{1}{S_{pi}^R} - n + 1.$$

Corollary 2.4. If all n nodes give the same absolute speedup S_p^A, then $S_p^A = S_p$. If all n nodes give the same relative speedup S_p^R, then

$$S_p^R \leq \frac{np}{np - p + 1} < \frac{n}{n - 1}.$$

Proof: The first assertion follows immediately from the above theorem, since $\sum_{i=1}^{n} g_i = 1$. For the second, we see that when the relative speedups are all equal

$$\frac{1}{S_p} = \sum_{i=1}^{n} \frac{1}{S_p^R} = n + 1 - \frac{n}{S_p^R} - n + 1.$$

Since $S_p \leq p$, this implies

$$\frac{n}{S_p^R} - n + 1 \geq \frac{1}{p} \text{ or}$$

$$S_p^R \leq \frac{np}{np - p + 1} = \frac{n}{n - 1 + \frac{1}{p}} < \frac{n}{n - 1}. \blacksquare$$

Each node of the graph can be an arbitrarily complex nested loop containing DOSERIAL, DOALL, and DOACR loops. The problem of optimal static processor allocation to such nodes has been solved optimally and is discussed in Chapter 4.

Corollary 2.4.1. A program can not be partitioned into n disjoint segments so that the relative speedups are 1, 2, ..., n respectively, for $n \geq 3$.

Proof

If there was a program with the above property, then from Theorem 2.2 we would have,

$$\frac{1}{S_p} = \sum_{i=1}^{n} \frac{1}{i} - (n - 1) = H_n - (n - 1)$$

where H_n is the n-th harmonic number and therefore,

$$S_p = \frac{1}{H_n - (n - 1)} < 0$$

because for $n \geq 3$ we have $H_n < (n - 1)$. ■

Corollary 2.4.2. If a program is partitioned into n disjoint segments with all segments having a relative speedup $S^R = \frac{1}{n} S_p$, and if $n \leq p - 1$, then

$$S_p = n + 1 \quad \text{and therefore} \quad S^R = \frac{n + 1}{n} \tag{2.4}$$

Proof: By substituting S^R in Theorem 2.2 we get,

$$\frac{1}{S_p} = \frac{n - (n - 1)S^R}{S^R} = \frac{n^2 - (n - 1)S_p}{S_p}$$

and after simplification we have $n^2 - (n - 1)S_p - 1 = 0$. Solving for S_p, we finally get $S_p = n + 1$. ■

Case 2. (Vertical parallelism). Let the task graph be *flat,* i.e. let there be a single layer V consisting of n nodes. This is the case when no dependencies exist between any pair of nodes. Here we may exploit vertical parallelism by executing all program nodes simultaneously. We consider the extreme case where each node requests exactly one processor. Since each node is allocated one processor, if $n \leq p$ the execution time is dominated by the largest task. In the general case bin-packing can be used to evenly distribute the n nodes into p bins. In Chapter 6 we discuss D&F, a heuristic algorithm for this case. This algorithm performs better than Multifit, the best known heuristic so far [CoGJ78]. Then, if b_i, $1 \leq i \leq p$ denotes the largest bin, we have the following theorem.

Theorem 2.3. The total speedup resulting from the parallel execution of an n-node flat graph on p processors, where each node is allocated one processor, is given by

$$S_p = \frac{1}{\sum_{v_j \in b_i} g_j}.$$

Proof: The proof follows directly from the definition of speedup and the assumptions stated above. ■

Corollary 2.5. If $n \leq p$ then

$$S_p = \frac{1}{\max(g_1, g_2, \ldots, g_n)}.$$

Proof: This follows from the previous theorem since each bin contains one node. ■

Case 3. (Horizontal and vertical parallelism). In the most general case we have a program that exhibits both types of parallelism, horizontal and vertical. In other words, the task graph consists of k (> 1) disjoint layers $V_1, V_2, \ldots, V_k$ with at least one layer containing two or more nodes (Figure 2.1). If $|V_i|$ is the cardinality of the i-th layer, we assume that $|V_i| \leq p$, ($i = 1, 2, \ldots, k$). (In the case of $|V_i| > p$ we fold and fuse nodes such that $|V_i| \leq p$ (Chapter 6).) Our aim in this case is to exploit horizontal and vertical parallelism in the best possible way. Maximizing speedup is equivalent to minimizing parallel execution time. For each node of the task graph v_j, we define r_j to be the maximum number of processors that the node could use. When $r_j = 1$, ($j = 1, 2, \ldots, n$) our problem is reduced to the classical multiprocessor scheduling problem, which has been proved NP-Complete [GaJo78]. Our general problem can be reduced to the latter one by decomposing each node v_j into r_j independent sub-nodes of equal size. This trivially proves that our problem is also NP-Complete. Heuristic solutions are therefore the only acceptable approach to solving the problem suboptimally in polynomial time.

In Chapter 6 we discuss an efficient linear-time heuristic algorithm for allocating processors to general task graphs. The total program speedup on p processors that results from the application of this heuristic is given by the following theorem (Chapter 6).

Theorem 2.4. The total program speedup that results from the parallel (vertical and horizontal) execution of a k-layer task graph on p processors is given by

$$S_p = \frac{1}{\sum_{i=1}^{k} \lambda_i} \quad \text{where} \quad \lambda_i = \max_{v_j \in V_i} \left\{ \frac{g_j}{S^A_{p_j}} \right\} \tag{2.5}$$

where S^A_{pj} is the absolute speedup of node v_j when p_j processors are allocated to it.

Note that Theorems 2.2 and 2.3 are special cases of Theorem 2.4. If the graph is reduced to a flat graph, then k=1 and Corollary 2.5 holds. On the other hand, if each layer contains one node (linearized) then $q_i = 1$ in (2.5), ($i = 1, 2, ..., k$), and thus Theorem 2.2 holds true.

2.5. Speedup and Processor Allocation for DOACR Loops

In this section we focus on a single node of the task flow graph representing the given program, i.e. a DOACR loop. We extend and generalize the do across model, to allow for idle processor time caused by do across delays. In [Cytr84] it is assumed that no processor may become idle unless it completes all iterations assigned to it. This is true only in certain special cases. As we shall see later by means of an example, each processor may have to idle between successive iterations. The following theorem generalizes the do across model and accounts for idle processor time.

Theorem 2.5. Consider a DOACR loop with N iterations and delay d, and let b denote the execution time of the loop-body. Then p processors can be allocated to the iterations of the loop in such a way that the speedup is given by $S_p = Nb/T_p$, where

DOACR $I = 1, 8 \quad \{d=4\}$

$\}$ 10

END

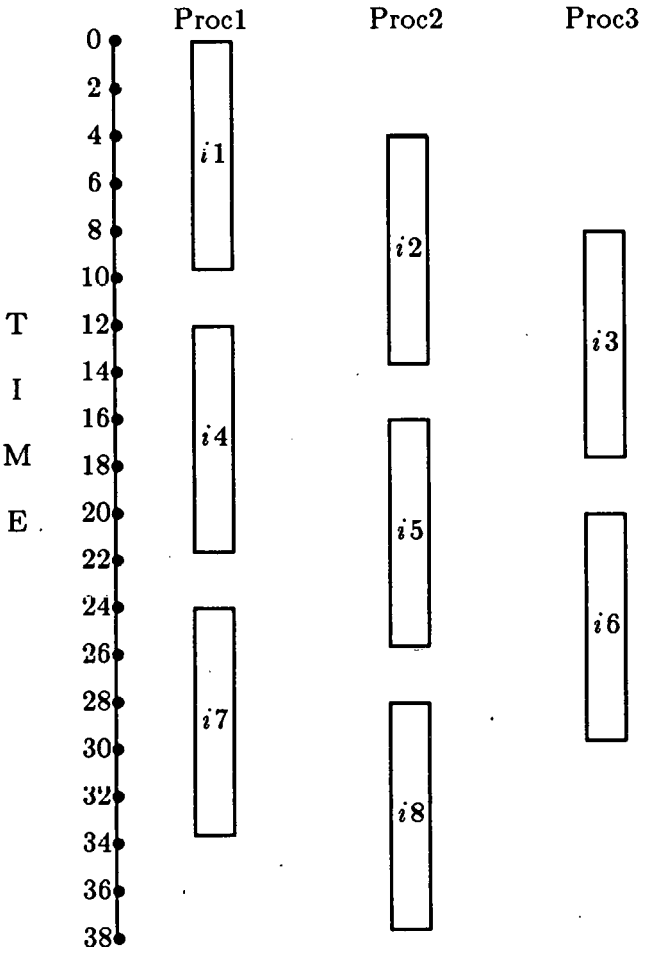


Figure 2.2. An example of the application of Theorem 2.5 for $p = 3$.

$$T_p = \left(\lceil N/p \rceil - 1\right) \max(\mathrm{b}, \mathrm{pd}) + d((N-1) \bmod p) + b \qquad (2.6)$$

Proof: Let us number the iterations 1, 2,..., N in their natural order and the processors

1, 2,..., p in any order. The processors are allocated to the iterations as follows. Assume first that $N > p$. Iteration 1 goes to processor 1, iteration 2 goes to processor 2, ..., iteration p goes to processor p. Then iteration $(p + 1)$ goes to processor 1, etc., and this scheme is repeated as many times as necessary until all the iterations are completed. We can now think of the iterations arranged in a $\lceil N/p \rceil \times p$ matrix, where the columns represent processors. If $N \leq p$, we employ the same scheme, but now we end up with a $1 \times N$ matrix instead.

Let t_k denote the starting time of iteration k, $(k = 1, 2,..., N)$, and assume that $t_1 = 0$. Let us find an expression for t_N. First, assume that $N > p$. For any iteration j in the first row, the time t_j is easily found:

$$t_j = (j - 1)d \qquad (j = 1, 2,..., p).$$

Now iteration $(p + 1)$ must wait until its processor (i.e., processor 1) has finished executing iteration 1, and d units of time have elapsed since t_p, the starting time of iteration p on processor p. Hence

$$t_{p+1} = \max(b, t_p + d) = \max(b, pd).$$

The process is now clear. If we move right horizontally (in the matrix of iterations), each step amounts to a time delay of d units. And if we move down vertically to the next row, each step adds a delay of $\max(b, pd)$ units. Thus the starting time t_k for an iteration that lies on row i and column j will be given by

$$t_k = (i - 1)\max(\mathrm{b}, \mathrm{pd}) + (j - 1)d.$$

For the last iteration, we have $i = \lceil N/p \rceil$ and

$$j = \begin{cases} N \bmod p & \text{if N mod p} > 0 \\ p & \text{otherwise} \end{cases}$$

Since $j - 1$ can be written as $(N - 1) \bmod p$, we get

$$t_N = \left(\lceil N/p \rceil - 1\right) \max(b, pd) + d((N-1) \bmod p).$$

Now let $N \leq p$. It is easily seen that

$$\begin{aligned} t_N &= (N-1)d \\ &= \left(\lceil N/p \rceil - 1\right) \max(b, pd) + d((N-1) \bmod p). \end{aligned}$$

Finally, since the parallel execution time T_p is given by $T_p = t_N + b$, the proof of the theorem is complete. ■

Figure 2.2 shows an example of the application of Theorem 2.5. The DOACR loop of Figure 2.2 has 8 iterations, a delay $d = 4$, and a loop-body size of 10. The total parallel execution time on $p = 3$ processors is 38 units, as predicted by Theorem 2.5. We can maximize the speedup of an arbitrarily nested DOACR loop which executes on p processors by using an optimal processor allocation algorithm described in Chapter 4.

Corollary 2.6. Consider a sequence of m perfectly nested DOALL loops numbered 1, 2,..., m from the outermost loop to the innermost. Let S_{p_i} denote the speedup of the construct on a p_i-processor machine when only the ith loop executes in parallel and all other loops serially, $(i = 1, 2, \ldots, m)$. Then the speedup S_p on a p-processor machine, where $p = \prod_{i=1}^{m} p_i$ and p_i processors are allocated to the ith loop, is given by $S_p = \prod_{i=1}^{m} S_{p_i}$.

2.6. Multiprocessors vs. Vector/Array Machines

In this section we consider an application of Lemma 2.1. Generally in a restructured program we have vector constructs that can execute in parallel on an SEA system. Let S_v denote the speedup that results by executing a program PROG, on an SEA machine. Obviously vector statements can be executed in parallel on any MES system as well (perhaps with a significantly

higher overhead). Do across loops with $d > 0$ can execute in parallel only on MES systems. Let S_m denote the additional speedup we achieve by executing the DOACR loops of PROG with $d > 0$ in parallel. Finally, let S_o denote the additional speedup that results by overlapping disjoint code modules during execution (vertical parallelism or high level spreading). If S_P is the total speedup we obtain by executing PROG on an MES machine, then from Lemma 2.1 we have

$$S_P = S_v * S_m * S_o$$

It is clear that for SEA systems we always have S_m=1 and $S_o = 1$, while for MES systems all three components may be greater than one. If $S_{SEA}(PROG)$ and $S_{MES}(PROG)$ denote the overall speedups of PROG for SEA and MES systems respectively, then

$$S_{SEA} = S_v \qquad \text{and}$$

$$S_{MES} = S_v * S_m * S_o$$

Assuming no overhead of any type or the same overhead for both systems, the S_v term should have a value that depends on the program characteristics (and is independent of the machine architecture). For each program we can therefore measure the additional speedup offered by MES systems. Let us define $\alpha(PROG)$, the MES *superiority index* as follows.

$$\alpha(PROG) = \frac{S_{MES}(PROG)}{S_{SEA}(PROG)} = S_m * S_o$$

where $1 \leq \alpha(PROG) \leq P$. α is computed on a program basis and can be used as a relative performance index for MES architectures (PROG can execute α times faster on an MES system than on an SEA machine).

CHAPTER 3

PROGRAM PARTITIONING AND INTERPROCESSOR COMMUNICATION

In this chapter we consider two closely related problems: Program partitioning and interprocessor communication. These are two popular terms in parallel processing, but there is no precise definition as to what their meaning and use are [GGKM83], [Ston77], [Veid85]. Intuitively these two terms are self-explanatory: Program partitioning refers to the process of breaking a program down to smaller components, but this can be done by using several different approaches and for different objectives. Questions like how we partition a program, what are the boundaries between partitions or parts, how we define a "part", what are the trade-offs and the precise goals of program partitioning remain largely unanswered. The term "interprocessor communication" is also self-explanatory. But similar questions about the precise meaning of interprocessor communication and its impact on program execution have no unique answers. Also, we do not have available a methodology for quantitatively characterizing these terms.

Here we attempt to define more precisely the problems of program partitioning and interprocessor communication, to model them, identify the variables involved and quantify them. We see below that program partitioning, interprocessor communication, parallelism, and data dependences are all closely related. Recall that our machine model is a *shared* memory parallel processor system as shown in Figure 1.1. The processor-memory interconnection network can be a multistage interconnection network or a bus. In Figure 1.1 we also have a dedicated processor-to-processor interconnection network. As usual we consider the case of the dedicated execution of a single parallel program. Data dependences were also defined in Chapter 1.

Let us consider initially the problem of interprocessor communication for the case of a 2-processor system. During the parallel execution of a program, different program modules will execute on different processors. Since both processors work on the same program there should be some coordination among them. One processor must "inform" the other at certain instances about specific events. The process of information exchange between two or more processors executing the same program is called *interprocessor communication.* We can distinguish two types of interprocessor communication: *Data communication,* during which one processor receives data that it needs from other processors, and *control communication,* where processors exchange control information, for example to announce an event or to coordinate execution. Data communication is mostly program dependent. Control communication depends highly on the architecture of the machine and is necessary only because it is needed to impose an order under which specific events must take place (e.g. order of execution). Both types of interprocessor communication are significant because both are reflected as overhead in the total execution time of a program.

Control communication is involved in activities such as barrier synchronization, semaphore updates, or invocations of the operating system. Control communication is usually involved in data communication as well, as shown later. For the most part in this chapter, we ignore control communication or assume that it takes a constant amount of time to check/update a semaphore, set a flag, or activate a process.

As mentioned in Chapter 1 from the DDG of a program we can derive different compact DDG's or CDDG's with nodes representing blocks of code (instead of statements) and arcs representing collections of dependences between nodes. Each different CDDG defines a different program partition. Later in the chapter we see how to construct the appropriate CDDG or program partition. Let $G(V, E)$ be a compact data dependence graph of a given program. This

directed graph $G(V,E)$ is called the *program task graph* or simply the *task graph*. The nodes V of G are the tasks of the program and arcs E represent data dependences among tasks. Each node $v_i \in V$ makes a request for r_i processors (i.e., task v_i can use at most r_i processors). We have serial and parallel tasks when $r_i = 1$ and $r_i \geq 1$ respectively. During program execution a parallel task spawns two or more processes. A parallel loop for example can be considered a parallel task with one or more iterations forming a process. A task graph may be connected or disconnected. The cardinality of G is the number of connected subgraphs in G. If $|G| = |V|$ then any parallel execution of G will involve an interprocessor communication of zero. The *indegree* of a node of G is the number of dependence arcs pointing to it, or equivalently, the number of immediate predecessors of that node. Similarly we define the *outdegree* of a node to be the number of immediate successors or the number of arcs originating from that node. The indegree of a subgraph G' of G is the number of arcs $u \rightarrow v$ of the form u does not belong to G' and $v \in G'$. The *outdegree* of G' is the number of arcs of G such that $u \rightarrow v$, $u \in G'$ and v does not belong to G'. A node or subgraph of G is said to be *ready* if its indegree is zero. Note that if we execute data independent program modules (i.e., with zero indegree) in parallel, data communication does not occur. All nodes of a subgraph with zero indegree can be executed in parallel. Consider the two loops of Figure 3.1. The second loop is data dependent on the first loop. However we can still execute them in parallel, each on a different processor assuming synchronized write/read access to array A by each processor.

3.1. Goals and Trade-offs

The goal of this chapter is to study the problem of minimizing overhead due to interprocessor communication. Communication overhead obviously occurs at run-time, but we want to deal with the problem at compile-time (the reason being that any extra run-time activity will incur additional overhead). We will develop techniques that will be applied to the source program at

```
DO   i=1,n
     . . .
S1:  A(i) = B(i) * C(i)
     . . .
ENDO

DO   i=1,n
     . . .
S2:  D(i) = A(i) + E(i)
     . . .
ENDO
```

Figure 3.1. Example of data communication.

compile-time and will result in the reduction of communication at run-time. Our main criterion is to design these schemes such that when applied to a given program, they do not reduce the degree of potential parallelism in that program. In other words, if $PROG_1$ and $PROG_2$ denote a parallel program before and after these schemes are applied, the execution time of $PROG_2$ on a p-processor machine under any scheduling policy will be less than or equal to the execution time of $PROG_1$ under the same scheduling policy.

Now let us see how communication takes place between two tasks u and v where v is data dependent on u. If during execution u and v run concurrently on different processors, data computed in u must be sent to v and the overhead involved is explicitly taken into account. Another alternative is to execute u to completion and thereafter execute v on the *same* processor(s). Thus data computed in u and used by v can reside in the corresponding processor(s), and therefore no explicit communication through the interconnection network is needed. In such a case the communication overhead would be zero. Yet another alternative is to

execute v after u has completed, and possibly on a different set of processors. In this case we assume the existence of *prefetching* capabilities in the system (e.g., Cedar machine), i.e., the data computed in u and used by v are written into the shared memory (upon completion of u) and prefetched (for example to registers) before v starts executing. The communication overhead in this case will also be zero. Note that this latter approach allows other tasks to execute between the time u completes and the time v starts. It also takes care of local memory limitations, e.g. when the processor(s) executing u is unable (due to memory limitations) to keep the data needed by v until v starts executing. In summary, we can not avoid overheads due to "real time" communication that occurs when tasks execute concurrently, but by using appropriate techniques we can eliminate or reduce this overhead when the tasks involved execute on different time intervals.

From the above discussion it becomes apparent that the approach we will use is to reduce interprocessor communication by disallowing high level spreading whenever appropriate. It is clear that by prohibiting high level spreading we reduce the degree of potential program parallelism. As shown later however, this is done only when it can be guaranteed that the savings in interprocessor communication outweight the potential loss of parallelism irrespectively of the scheduling scheme used.

In what follows when we explicitly prohibit two or more (data dependent) tasks from executing concurrently, we say that these tasks are *merged*. Thus task merging does not lexically merge tasks but it implies that the merged tasks can execute on the same or different sets of processors, but on different time intervals. This restriction can be relaxed for certain cases as shown later. For the purpose of this work we assume that merged tasks execute on adjacent time intervals (one after the other), and the data computed by one task reside in local memory until the successor task can use them. During execution each task is assigned a set of processors which

remains fixed throughout execution of that task, or varies dynamically during execution.

The process of merging defines a *partition* of the program into disjoint code modules or tasks. Program partitioning is the end-product of minimizing communication overhead. We wish to have a structure that represents a program without "hiding" any of its parallelism. This structure should be a low-level representation of the program, and in our case the best one is the data dependence graph (DDG). Any other higher level program graph can be derived from the DDG by merging together some of its nodes. Merging however may hide some of the parallelism inherent in the DDG. The extreme case is considering the entire program (DDG) as a single node (task). (It should be emphasized that we are not concerned about how to schedule a program graph at this point; and the material of this chapter is applicable to any scheduling scheme.) Since merging reduces the degree of parallelism, it should be done only when it can be proved that the (resulting) reduced graph will have a shorter execution time than that of the previous graph, under any scheduling scheme. Clearly independent tasks can never be merged, unless scheduling overhead is taken into account and that is the subject of Chapter 7.

In the following sections we propose optimal and near-optimal solutions to the problems of program partitioning and minimization of interprocessor communication. First we consider an idealized program model consisting of atomic operations, and without coarse grain constructs. All atomic operations have equal execution times. Even though this is not a very realistic program model, a few functional programs fall into this category. We propose a model for quantifying interprocessor communication for such programs, and present an algorithm that generates optimal partitions for chains of tasks. This algorithm also generates optimal time/communication schedules for this particular case.

Then we consider the case of real Fortran programs and show how some of the above ideas can be applied to real programs. The algorithm mentioned above can be used to obtain optimal

partitions of straight-line code in Fortran programs, chains of serial loops, and some types of parallel loops. However the idiosyncrasies of Fortran do not permit a direct mapping to the idealized model. Furthermore, these idiosyncrasies can be used to our advantage to obtain fast, efficient partitions of real programs heuristically. These partitions are locally optimal and are appropriate for efficient execution on real parallel machines.

3.2. More on Communication and Partitioning

For the rest of this chapter interprocessor communication refers to data communication alone, unless stated otherwise. Let u_i denote the task (program module) assigned to processor i, in a parallel processor system with p processors. Then interprocessor communication, or data communication from processor i to processor j, takes place if and only if u_j is data or flow dependent on u_i, or when data computed in u_i are used by u_j. Note that anti-dependences and output dependences are satisfied through control communication only. Therefore data communication refers to interprocessor communication that involves explicit transmission of data between processors. We use $S_j(i)$ to denote the amount of data *sent* by processor i to processor j. Similarly $R_j(i)$ denotes the amount of data *received* by processor j from processor i. For each task, $R_i(j) \equiv S_j(i)$. If $\{i_1,\ldots, i_k\}$ is a set of tasks that execute in parallel, then

$$R_j(i_1,i_2,\ldots,i_k) \equiv S_j(i_1,i_2,\ldots,i_k) = \sum_{m=1}^{k} S_j(i_m)$$

Let us define the *unit of data* to be the value of a scalar, and the *communication unit* τ to be the time it takes to transmit a unit of data between two processors. Then the time spent for communication during the concurrent execution of the two loops of Figure 3.1 on two processors would be τ^*n.

Interprocessor communication takes place only during the parallel execution of data dependent tasks. By executing such tasks in parallel we may reduce the execution time if the amount of communication is not too high. Clearly there is a tradeoff between parallelism and communication. Communication is minimized when all tasks execute on the same processor; parallelism however is also minimized in such a case. Parallelism on the other hand tends to be maximized (or equivalently execution time tends to be minimized) when each task executes on a different processor. This however maximizes interprocessor communication. The problem of simultaneously maximizing parallelism and minimizing communication is a hard optimization problem that has been proved NP-Complete [Ston77], [GaJo79]. Communication takes place to satisfy data dependences. Another way of viewing this relation is the following: communication quantifies the notion of flow dependence between different tasks.

We can intuitively define the *degree of parallelism* as the number of ready tasks at any given moment. Obviously in a system with p processors we want the degree of parallelism to always be at least p. In general, the degree of parallelism and interprocessor communication are incompatible. The goal during program partitioning is to decompose the program as much as possible to keep the degree of parallelism close to p, and at the same time to have as many independent tasks as possible to keep interprocessor communication low. Usually it is impossible to optimize both objectives since optimizing one counteroptimizes the other. For certain cases however there is an "equilibrium" point that minimizes the parallel execution time.

In the following sections we develop a formal model for interprocessor communication and study the impact of data communication on execution time. For special types of program graphs we can obtain optimal partitions for an unlimited number of processors, and optimal processor allocations for a limited number of processors, where execution time is minimized also taking into account interprocessor communication. Then we consider program partitioning for the

specific case of Fortran programs and show how the above model can be applied to minimize data communication between certain types of Fortran tasks.

3.3. Methods for Program-Partitioning

There are two approaches for partitioning a program. The *top-down* approach starts with a single task which is the entire program. Then following some rules it decomposes the program into smaller tasks in a recursive way. The second is a *bottom-up* approach. Starting with the lowest level graph representation of the program, it tries to merge nodes together to form larger tasks. The best low level representation of a program is the DDG. In both cases we have to split or merge nodes repeatedly until 1) we have enough tasks to assign to p processors at each moment during execution, and 2) the splitting/merging creates tasks that are "as independent as possible". Heuristic algorithms could be used to obtain a suitable partition. In terms of available information, the bottom-up composition is superior to the top-down decomposition. This is true since during decomposition of the entire program we do not have information about its internal structure, and extensive searching must take place. During composition however, we have information about the global structure of the program (that can be easily maintained by the compiler) as well as about its basic components. We can therefore perform local optimizations that may be impossible or very expensive to do in the top-down approach. For what follows we assume that program partitioning is performed through composition starting from the data dependence graph of the program.

Let $G_i \equiv G_i(V_i,E_i)$ be a directed graph. Our approach to program partitioning is to start from G_1, the DDG of the program, and through a series of transformations that create a sequence of CDDGs $G_2,G_3,\ldots,G_{k-1}$, construct G_k which gives the program task graph and therefore the final partition. Since the construction of the program graph is done through composition

we have $|V_1| \leq |V_2| \leq \cdots \leq |V_k|$ and $|E_1| \leq \cdots \leq |E_k|$. G_k will have enough tasks to keep all p processors busy while minimizing interprocessor communication. Our approach to program partitioning involves the design of exact and approximation algorithms that accomplish the following goals.

- Tasks are merged together if and only if the resulting graph is better (in terms of execution time) under any circumstances, i.e., under any scheduling scheme and for any number of processors.
- Optimal partitions for special types of graphs with serial nodes.
- Optimal partitions for special types of graphs with serial and parallel nodes and for an unlimited number of processors.
- Near-optimal partitions for general task graphs.
- Efficient near-optimal partitions for task graphs of real Fortran programs.

Nodes are merged only if data communication is very high. In this way we reduce only the necessary degree of parallelism to keep communication at tolerable levels. The next section describes the model that can be used to construct G_k.

3.4. A Model for Quantifying Communication

Consider a machine with p processors that are connected through a bus to a shared memory. The model starts with a representation of the program as a directed task graph as defined in the previous section. Candidates for merging are nodes whose possible parallel execution involves a large amount of data communication.

Let G_1 be the DDG of a program. The first composition of tasks finds all strongly connected components of G_1. Each strongly connected component forms a task and let G_2 be the resulting task graph. The arcs of G_1 and G_2 are labeled with weights called the *communication traffic* or *weight* such that the weight w_i of arc $e_i=(v_i,v_j)$ is given by

$$w_i = \tau^* m$$

where m is the number of data items that need to be transmitted from task v_i to task v_j, and τ is the communication constant. For example the communication traffic between the two loops of Figure 3.1 is $\tau^* n$. During the reduction of G_1 to G_2 the composition of new tasks is performed as follows:

Let $V_1 = \{u_1, u_2, \ldots, u_n\}$ be the tasks of G_1 and t_{v_i}, $(i = 1, \cdots, n)$ be their serial execution times. Suppose now that subgraph $H_i \subseteq G_1$ is a strongly connected component of G_1. Then H_i is replaced by a node u^H with execution time

$$t_i = \sum_{v \in H_i} t_v.$$

Arcs are merged using the following procedure: For each task v not in H_i, replace all arcs $e_1^v, e_2^v, \ldots, e_j^v$ originating from v and such that $e_i^v = (v, u_k)$, and $u_k \in H_i$, $(i = 1,2,\ldots,j)$ with an arc $e^v = (v, u^H)$ which has a weight w^u given by

$$w^u = \sum_{i=1}^{j} w_i^v.$$

After the first reduction the resulting graph G_2 is a connected or disconnected graph without nontrivial strongly connected components. Therefore G_2 is a directed acyclic graph or DAG.

During parallel execution of G_2, two (or more) independent or data dependent nodes may execute simultaneously. Let $u_i \rightarrow u_j$ be two data dependent tasks. If u_i and u_j execute simultaneously on two processors, the total execution time T will be

$$T = \max(T_1^i, T_1^j) + w_{ij}$$

This is a reasonable assumption since the processors are connected through a bus and bus transactions are serial. The communication time w_{ij} is reflected in the total execution time since the processor executing u_i incurs an overhead to transmit the data, and the processor executing u_j must wait until the data arrives.

Let us assume that the total overhead is equal to the time it takes to transmit the data. Therefore if tasks u_i and u_j execute on the same processor, the communication can be done through the local memory inside each processor and it is ignored. The execution time in that case will be

$$T_1 = T_1^i + T_1^j.$$

We also assume that tasks are collections of atomic operations, and if u is a parallel task and T_1^u is its serial execution time, then the parallel execution time T_p^u of u on $p \leq T_1^u$ processors will be

$$T_p^u = \frac{T_1^u}{p}. \tag{3.1}$$

Consider a set of two tasks $\{u, v\}$. We use the notation $(set\text{-}x/task\text{-}x)$ to describe the execution of u and v, where x can be *serial* or *parallel*. *Task-serial* (or t-s) means that both u and v execute serially. *Task-parallel* (or t-p) indicates that at least one of u and v executes on more than one processor. *Set-serial* (or s-s) means that both u and v execute on the same *set* of processors (perhaps in parallel). If both execute on p processors, then u will complete execution before v starts. If u executes on p and v on p' processors and $p < p'$, then p' processors are given to the set and u and v start simultaneous execution with u executing on the first p processors and v executing on the remaining $p'-p$ processors. When u finishes executing, all p' processors are taken over by v. *Set-parallel* (or s-p) means that u and v are executed concurrently on disjoint sets of processors. Therefore there are four possible ways of executing u and v. $(s\text{-}s/t\text{-}s)$ describes the case where both u and v execute serially on the same processor. $(s\text{-}s/t\text{-}p)$ when each of u and v executes in parallel but on the same set of processors. $(s\text{-}p/t\text{-}s)$ is the case of u and v executing serially but each on a different processor. Finally $(s\text{-}p/t\text{-}p)$ denotes the case where each of u and v executes in parallel and both execute con-

currently on disjoint sets of processors. The above notation can also describe the execution mode for sets with an arbitrary number of tasks.

As mentioned above, interprocessor communication is ignored for the case of $(s\text{-}s/t\text{-}s)$ or $(s\text{-}s/t\text{-}p)$ when all tasks execute on the same number of processors. For the case of two tasks u and v the total execution time is defined as follows:

$$(s\text{-}s/t\text{-}s) \rightarrow T_1^u + T_1^v \tag{3.2}$$

$$(s\text{-}s/t\text{-}p) \rightarrow T_p^u + T_p^v \tag{3.3}$$

(if both execute on the same p processors)

$$(s\text{-}p/t\text{-}s) \rightarrow \max(T_1^u, T_1^v) + w_e \tag{3.4}$$

$$(s\text{-}p/t\text{-}p) \rightarrow \max(T_p^u, T_{p'}^v) + w_e \tag{3.5}$$

where w_e is the weight of the arc $e=(u,v)$ (if any) and p, p' are disjoint sets of processors used by u and v, respectively.

Note that for the $(s\text{-}s/t\text{-}p)$ case, if u and v execute on a different number (but the same common subset) of processors, say p and p' respectively, then the total execution time is defined as follows: Suppose $p < p'$, i.e., both tasks will execute on a common set of p processors but task v will use an extra $p' - p$ processors. Suppose also that task v will finish execution after task u has completed. Both tasks start executing concurrently, u on p processors and v on $p'\text{-}p$ processors. When u terminates, task v takes over the remaining p processors and executes on p' processors until it completes. Since the overlap (i.e. set serialization) is not perfect some interprocessor communication will occur in this case. We assume that the data communication is proportional to $p'\text{-}p/p'$. In other words if the total amount of data communication from u to v is w, the amount of data that must be explicitly transmitted will be inversely proportional to the the number of common processors. In the above case the data communication that must be transmitted through the bus will be $(p'\text{-}p/p')w$. When $p'=p$ (total set serialization), the

communication is zero. When p' and p are disjoint sets of processors (set parallel), then $p=0$ and therefore the data communication is w as would be expected based on the previous definitions. Therefore the total execution time for the case of $(s\text{-}s/t\text{-}p)$ where u and v execute on p and p' processors respectively is

$$t_{u,v} = T_p^u + \frac{T_1^v - (p' - p)T_p^u}{p'} + \left\lceil \frac{p' - p}{p'} \right\rceil w. \tag{3.6}$$

Since this case is not truly set-serial let us denote it with $(s\text{-}\tilde{s}/t\text{-}p)$. Again the above can be easily extended for any set of tasks. We can say that the above notation describes four basic *schedules*. We can compare basic schedules, i.e., the corresponding schedule lengths (or execution times) using the following notation.

$$(s\text{-}?/t\text{-}?) \bullet (s\text{-}?/t\text{-}?)$$

where $? \in \{serial, parallel\}$ and $\bullet \in \{ <, \lesssim >, \gtrsim, =, \neq \}$. We can augment the basic schedule notation with a tuple that specifies the number of processors assigned to each task. For the case of the previous example $(s\text{-}s/t\text{-}p)\ (p,p)$, indicates that u will execute on p processors followed by the execution of v on the same p processors. The schedule corresponding to (3.6) can be uniquely characterized by $(s\text{-}\tilde{s}/t\text{-}p)\ (p,p')$. We can now state the following lemma.

Lemma 3.1 If u and v are two adjacent tasks connected by $e=(u,v)$ with a communication weight of w_e, and

$$(s\text{-}s/t\text{-}s) \lesssim (s\text{-}p/t\text{-}s) \quad \text{or} \quad T_1^u + T_1^v \leq \max(T_1^u, T_1^v) + w_e \tag{3.7}$$

then it is also true that

$$(s\text{-}s/t\text{-}p) \lesssim (s\text{-}p/t\text{-}p) \quad \text{or} \quad T_p^u + T_p^v \leq \max(T_p^u, T_p^v) + w_e. \tag{3.8}$$

Equivalently if the execution time of u and v when they execute serially on the same processor is smaller than when they execute simultaneously on two different processors, then the combined parallel execution time when they execute on the same set of p processors is smaller than when

each executes on its own set of p processors and their execution is concurrent.

Proof Since (3.7) is true, it follows directly that

$$w_e \geq \min(T_1^u, T_1^v).$$

Since by definition $T_p^u \leq T_1^v$ and $T_p^v \leq T_1^v$ we have

$$w_e \geq \min(T_p^u, T_p^v) \tag{3.9}$$

and by adding the same term to both sides of (3.9) we get

$$w_e + \max(T_p^u, T_p^v) \geq \min(T_p^u, T_p^v) + \max(T_p^u, T_p^v). \tag{3.10}$$

But the right handside of (3.10) is by definition equal to $T_p^u + T_p^v$, therefore (3.8) is also true. ■

Theorem 3.1 For a set of tasks $\{u, v\}$ and an unlimited number of processors, the following is a generalization of Lemma 3.1.

$$\text{if} \quad (s-s/t-s) \leq (s-p/t-s)$$

$$\text{then} \quad (s-\tilde{s}/t-p)(p, p') \leq (s-p/t-p)(p, p')$$

or equivalently (from (3.2), (3.4), (3.5), and (3.6))

$$\text{if} \quad T_1^u + T_1^v \leq \max(T_1^u, T_1^v) + w_e \tag{3.11}$$

$$\text{then} \quad T_p^u + \frac{T_1^v - (p' - p)T_p^u}{p'} \leq \max(T_p^u, T_{p'}^v) + \frac{p}{p'} w_e. \tag{3.12}$$

Proof We have two cases depending on which of u and v is larger.

Case 1: $T_1^u \geq T_1^v$, (i.e., task u is greater than or equal to task v). Then from (3.11) we have

$$T_1^v \leq w_e. \tag{3.13}$$

From (3.12) and (3.1) we have,

$$pT_1^v - \left(\frac{p' - p}{p}\right) T_1^u \leq pT_1^v \leq pw_e$$

which is true due to (3.13).

Case 2: $T_1^u < T_1^v$ (i.e., task u is smaller than v). Therefore from (3.11) we have

$$T_1^u \leq w_e. \tag{3.14}$$

Here there are two subcases.

Subcase 2.1: $T_{p'}^v \geq T_p^u$. (3.15)

Then after we carry out the calculations in (3.12) we have

$$pT_p^u \leq pw_e \quad \text{or} \quad T_p^u \leq w_e$$

which is (3.15) and therefore true.

Subcase 2.2: $T_{p'}^v < T_p^u$. (3.16)

Again from (3.12) we have

$$T_1^v - (p' - p)T_p^u \leq p_{w_e} \quad \text{or,} \quad pT_1^v - (p' - p)T_1^u \leq p^2 w_e$$

and finally

$$\frac{T_1^v}{T_1^u} \leq pw_e + \frac{p' - p}{p}. \tag{3.17}$$

But from (3.16) we have

$$\frac{T_1^v}{p'} \leq \frac{T_1^u}{p} \quad \text{or,} \quad \frac{T_1^v}{T_1^u} \leq \frac{p'}{p} \tag{3.18}$$

and therefore to show (3.17) it is enough to show that

$$\frac{p'}{p} \leq p^2 w_e + \frac{p' - p}{p}, \quad \text{or,} \quad w_e \geq \frac{1}{p}$$

which is true since $w_e \geq T_i^u > 1/p$. ■

Lemma 3.1 and Theorem 3.1 can be used to partition a program in a bottom-up approach, starting from its DDG representation and composing larger tasks by merging nodes of the DDG together whenever appropriate. But task merging reduces the possibilities for high level spreading and therefore the degree of parallelism. However by using Lemma 3.1 and Theorem 3.1 we can merge only those tasks that do not affect the degree of parallelism.

More precisely if G is a program task graph and u, v are tasks in G, then let G' be the graph derived from G by merging nodes u and v into a single node w. The merging of u and v takes place if and only if the execution time of G' is less than or equal to the execution time of G under any scheduling scheme and any number of processors. In other words we merge tasks together only when it is "safe" under any circumstances. More elaborate partitions may follow if necessary as described in the following sections.

We can merge tasks of a graph G in any order by checking repeatedly pairs of tasks in G. If a pair of tasks satisfies the conditions of Lemma 3.1 or Theorem 3.1, the two nodes in the pair are merged and form a single task. If for a given task there is more than one adjacent task for which the conditions are met, the order of merging becomes significant. Only in special cases we can find the optimal order and thus the optimal partition as explained below. However any merging that is based on the above tests is bound to reduce G into a task graph G' that will have an execution time less than or equal to that of G, irrespectively of the scheduling scheme used. For example an initial partition of the graph in Figure 3.2a will be formed by merging the first three nodes labeled 5, 10, and 15 into a single node labeled 30. This will be accomplished in two steps. The same partition will be obtained if we start from the leftmost or rightmost node.

For special types of graphs the optimal merging can be found. In addition the optimal schedule that minimizes execution time (taking into account interprocessor communication) can also be obtained for a limited number of processors.

3.5. Optimal Task Composition for Task Chains

In this section we present the *task composition* or TACOM algorithm that finds the optimal task composition for chains with data dependent serial tasks. For a limited number of processors, the algorithm also finds the optimal processor allocation for task chains that minim-

ize overall execution time including interprocessor communication. Lemma 3.1 and Theorem 3.1 are implicitly used to drive the algorithm. As shown by Theorem 3.1, if two tasks involve a large amount of data communication when they execute concurrently (but each serially), then they are combined to form larger tasks. The components inside these tasks can execute in parallel, but on the same physical processors. For this case we assume an idealized model where intertask communication is constant and thus independent of the way we partition and schedule a chain of tasks. We also assume that the data can be sent from one task to another at any time after the tasks start executing. This simplification is not very realistic but it makes it easier to describe the model and the algorithm. As it will be shown later, intertask (or interprocessor) communication changes depending on how tasks are grouped together and how they are scheduled. Assuming that the compiler can be used to evaluate intertask communication for each given configuration during the application of TACOM, precisely the same algorithm can be used to perform optimal partitioning and scheduling of real Fortran programs. Before we describe the algorithm let us define the necessary terms.

A *chain* graph is a directed graph of the type shown in Figure 3.2a, with $V=\{1,2,...,k\}$ and $E=\{e_i=(i,i+1) \mid i=1,...,k-1\}$. Each node i is associated with a weight t_i which is its serial execution time. Each arc $e_i=(i,i+1)$ is labeled with a weight w_{e_i} which gives the amount of communication traffic from node i to node $i+1$. A *fork-join* or FJ-graph is a graph of the type shown in Figure 3.2b consisting of a single source (node with indegree of zero), a single sink (node with outdegree of zero), and an arbitrary set of source-to-sink disjoint paths. The TACOM algorithm can perform optimal task composition for chain and FG-graphs. In what follows we consider the case of chains and the extension to FG-graphs is straightforward.

Consider a chain graph G, and let v_o and v_k be its first (source) and last (sink) node respectively. We can use the results of Lemma 3.1 and Theorem 3.1 to merge tasks that involve heavy

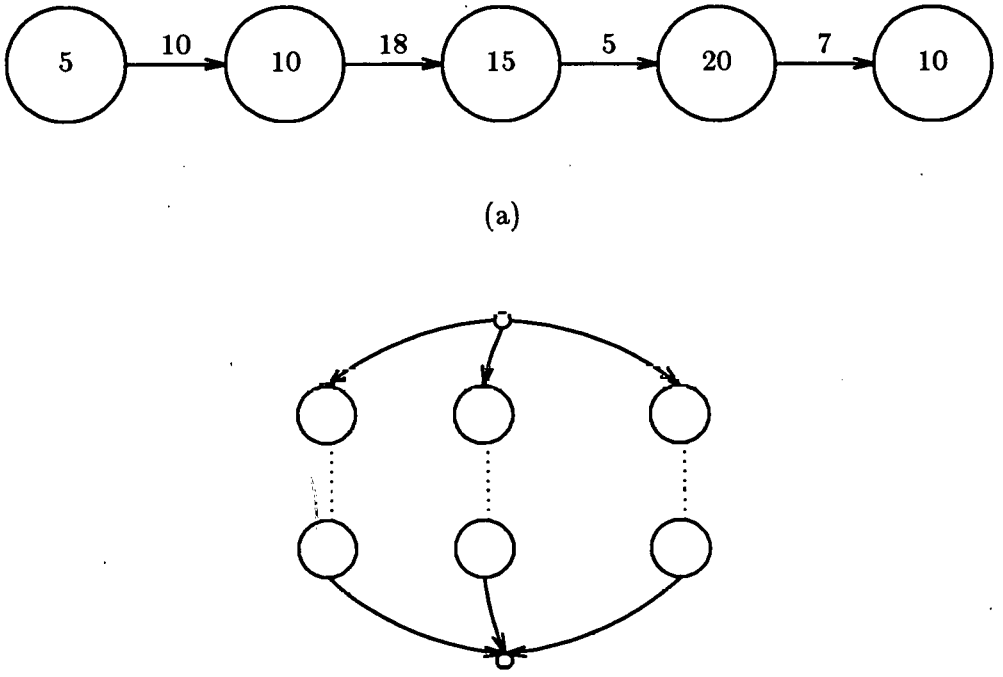


Figure 3.2. Example of chain and FJ-graphs.

data communication. However Lemma 3.1 gives us the necessary condition for merging two tasks but it does not give us any insight as to how the optimal merging can be achieved, in case of an arbitrarily long chain of tasks. The TACOM algorithm of which a similar version is given in [Poly84] and [Bokh85] uses Lemma 3.1 to choose pairs of tasks that are candidates for merging, and finds the optimal merging pattern as described below.

Let $V_G = \{1, 2,..., k\}$ be the tasks in chain G. We construct a layered graph L_G consisting of k layers $L_1, L_2, \ldots, L_k$. Nodes in L_G are represented by ordered pairs (i, j) such that $1 \leq i, j \leq k$ and $i \leq j$. A node (i, j) denotes the merging of nodes i through j (inclusive) into a single node. Each node (i, j) in L_G is labeled with t_{ij} defined by

$$t_{ij} = \sum_{m=i}^{j} t_m \tag{3.19}$$

where t_m is the label (serial execution time) of node m in G. The layers of L_G are constructed as follows. $L_1 = \{(1, j) \mid j = 1, 2,..., k\}$, i.e., L_1 contains the nodes corresponding to all combinations of merging tasks 1 through j, for $(j = 1, 2,..., k)$. There are k such nodes. Then for $(i = 2, 3,..., k)$ we construct $k - 1$ layers L_i which are defined as follows.

$$L_i = \left\{ (i, i), (i, i+1), (i, i+2),..., (i, k), (i+1, i+1),..., (i+1, k),...,(k-1, k), (k, k) \right\}.$$

The leftmost and rightmost nodes of L_i (with the exception of L_1) are (i, i) and (k, k) respectively. The L_G for the example of Figure 3.2a is shown in Figure 3.3. The ordered pair (i,j) at the left hand side of each node in Figure 3.3 denotes the tasks included (merged) into that node (i.e., tasks i through j inclusive).

Arcs in L_G exist only between successive layers and only connect nodes (i,j) and (m,l) such that $i \leq j$, $m \leq l$ and $m = j+1$. In layer L_i the first node is (i,i) and the number of different merging combinations that start from node i is $k-i+1$. Similarly all merging combinations that start from node $(i+1)$ are $k-(i+1)+1$. In general, the number of nodes in layer L_i, $(i=2,3,...,k)$ is

$$|L_i| = (k-i+1) + (k-(i+1)+1) + (k-(i+2)+1) + \cdots + (k-(k-1)+1) + (k-k+1)$$

or

$$|L_i| = \frac{(k-i+1)*(k-i+2)}{2}.$$

For the first layer of L_G we have $|L_1|=k$, since the first layer consists of all nodes $(1,i)$, for $(i=1,2,...,k)$. The outdegree of node (1,1) is $k-1$, of node (2,2) $k-2$, and in general the outdegree of node (i,i) is $k-i$. Note that in each layer L_i the node with the largest outdegree is (i,i). In fact if the nodes inside each L_i are in order as shown in Figure 3.3, then the outdegree of each

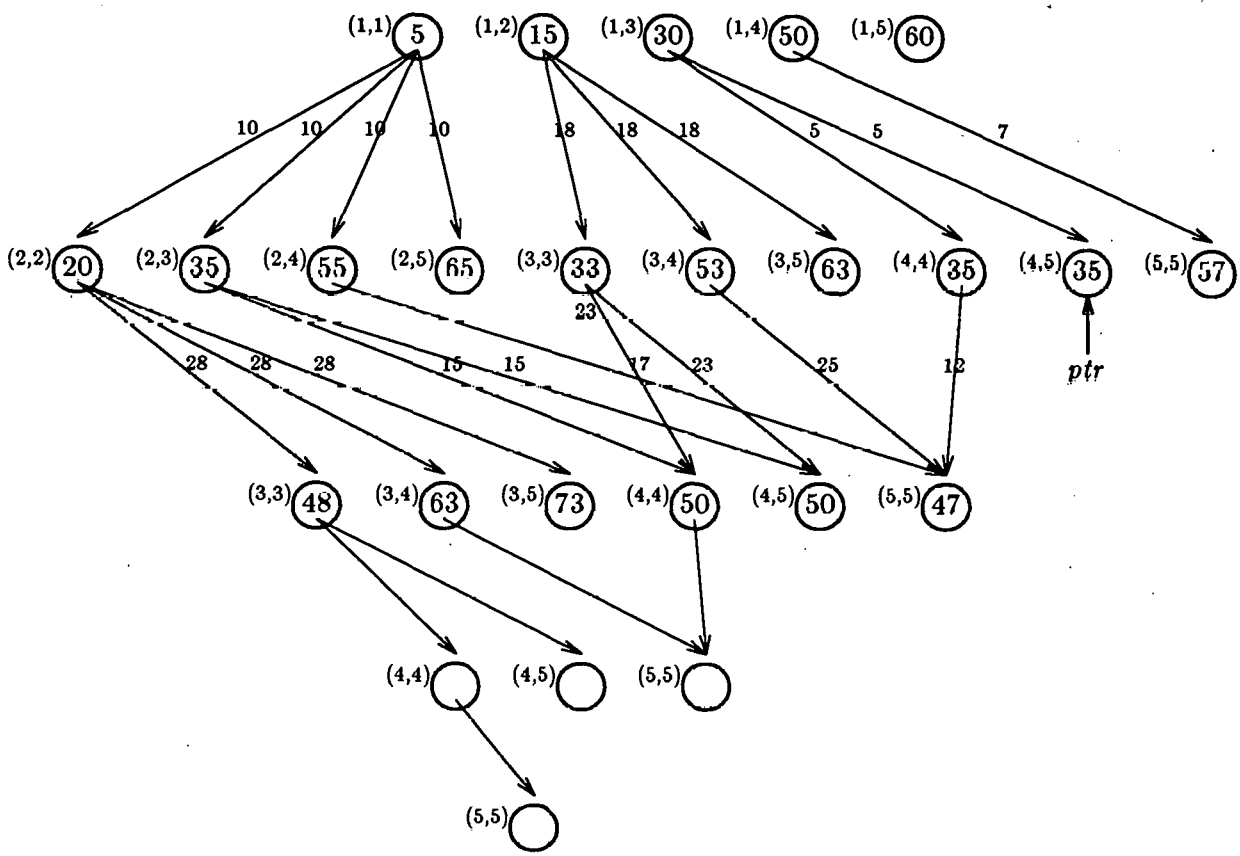


Figure 3.3. The application of the TACOM algorithm on the example of Figure 2a.

node from left to right is $[(k-i),(k-i-1),(k-i-2),\ldots,1,0]$, $[(k-i-1),(k-i-2),\ldots,1,0]$, ... , [1,0],[0].

The nodes of the first layer L_1 are labeled according to (3.19). The arcs originating from the tasks of L_1 are labeled as follows: From each node $(1,i)$ we have $(k-i)$ arcs originating from it and pointing to nodes of L_2 of the form $(i+1,j)$, for $i < j$. All these arcs are labeled with w_{e_i}, i.e., the communication weight of arc $e_i=(i,i+1)$ in G. This is repeated for $(i=1,2,\ldots,k)$ of L_1.

Then the algorithm relabels all nodes and arcs of L_G starting from L_2. In general, if the nodes of and arcs originating from layers $L_1,\ldots,L_{r-1}$ have been labeled, the nodes of layer L_r, $(r < k)$ are labeled as follows. Let (m, i) be a node of L_r connected to a number of nodes of the form $(h_r, m-1)$ of L_{r-1} (for $h_r \leq m-1 < m \leq i$ and all h_r). Let x_{h_r} be the label of $(h_r, m-1)$ and w_{h_r} be the weight of arc $(h_r, m-1) \rightarrow (m, i)$. Also let t_{mi} be defined by (3.19). The label x_m of node (m, i) is chosen from the labels of all nodes $(h_r, m-1)$ of L_{r-1} pointing to (m, i) as follows.

$$x_m = \min_r \left\{ w_{h_r} + \max(x_{h_r}, t_{mi}) \right\}. \tag{3.20}$$

The arc from which the node (m, i) was labeled is marked.

The arcs are labeled with the corresponding accumulated data communication weights as follows. Let $(m, i) \in L_r$ and e_1 be the arc $(h_r, m-1) \rightarrow (m, i)$ from which node (m, i) was labeled in the previous paragraph. Now let e_2 be the arc connecting (m, i) to a node $(i+1, g)$ of L_{r+1} (for $r < g$). Also let w_i be the weight of arc $e_i = (i, i+1)$ in the original chain G. Then the weight of e_2 is w_{e_2} and given by

$$w_{e_2} = w_i + w_{e_1}. \tag{3.21}$$

A global pointer *ptr* is maintained which at any given moment points to a node with the minimum label. The algorithm terminates when one of the following two conditions is met.

- All nodes of L_G are labeled, or
- All nodes at a given layer L_i have labels greater than the node pointed to by *ptr*, and *ptr* points to a node (j,k) of a previous layer.

After the algorithm terminates, the optimal merging of the tasks in G is given by following backwards the marked arcs starting from the node pointed to by *ptr*. The nodes in this path are of the type $(1, i_1), (i_1+1, i_2),\ldots, (i_m+1, k)$ where $1 \leq i_1 < i_2 < i_3 < \cdots < i_m < k$, and they

Input: A chain graph $G(V, E)$, with $V = \{1, 2,..., k\}$, $E = \{e_i = (i, i+1) \mid i = 1, 2,..., k-1\}$, task execution times t_i, $(i = 1, 2,..., k)$ and communication (arc) weights w_{e_i}, $(i = 1, 2,..., k-1)$.

Output: The optimal task composition (partition), or the optimal schedule of G on $p \leq k$ processors.

Method:

- From G construct a layered graph L_G with k layers $(L_1, L_2, \ldots, L_k)$. Nodes in layers are ordered pairs (i, j) such that $i \leq j$ representing the merging of nodes i through j of G. Layers are defined by the following sets:

$$L_1 = \{ (1, i) \mid i = 1, 2,..., k \}$$

$$L_l = \{ (m, i) \mid \text{such that } m \leq i,\ m = l, l+1,..., k, \text{ and } i = l, l+1,..., k \}$$

$$L_k = \{ (k, k) \}.$$

- FOR (all nodes (i, j), $i \leq j$, $i, j = 1, 2,..., k$ of L_G) DO
  - label (i, j) with $x_i = \sum_{m=i} t_i$

  ENDFOR

- FOR (all nodes $(1, i)$ of L_1) DO
  - label all arcs originating from $(1, i)$ with w_{e_i}.
  - ptr points to node $(1, i)$ with minimum x_i.

  ENDFOR

- FOR ($r = 2$, to $(k - 1)$) DO
  FOR (every node (m, i) in L_r) DO
  - By searching all nodes $(h, m-1)$ of L_{r-1} connected to (m, i) through e_h compute the label x_m of (m, i):

$$x_m = \min_h \{ w_{e_h} + \max(x_h, t_{mi}) \}$$

  - Mark the arc e_h corresponding to the minimum value, and label all arcs originating from (m, i) with

$$w_{e_h} + w_{e_{i+1}}$$

    where $w_{e_{i+1}}$ it the weight of $e_{i+1} = (i+1, j)$ in G
  - If $ptr > x_m$ then $ptr \leftarrow x_m$

  ENDFOR

  ENDFOR

- label node (k, k) of L_G with t_k.
- Reconstruct the optimal solution from ptr.

Figure 3.4. The task composition algorithm (TACOM).

uniquely define an optimal solution. A procedural description of the TACOM algorithm is given in Figure 3.4.

The application of this algorithm for the example of Figure 3.2a is shown in Figure 3.3. Note that the algorithm terminated when all labels of the third layer were found to be greater than that of *ptr*. The optimal solution for this example is (1,3)(4,5), i.e., tasks 1, 2, and 3 form a new task and tasks 4 and 5 form another composite task.

The construction of the layered graph L_G and the application of TACOM can be performed simultaneously. Each layer in L_G has at most $O(k^2)$ nodes and each node has an outdegree of at most $O(k)$. Since there are k layers in L_G the worst case performance of the algorithm is $O(k^4)$ and the average complexity is $O(k^3)$ where k is the number of tasks in the original chain G.

This algorithm can also be used to find the optimal schedule of a set of k data dependent tasks on p processors. In such a case the layered graph will consist of p layers and the path starting from *ptr* will give us the schedule that minimizes parallel execution time taking into account interprocessor communication cost.

3.5.1. Task Chains with Serial and Parallel Tasks

The same basic algorithm can be used to generate optimal partitions for the case of chain graphs with parallel and serial tasks that are to be executed on a system with an unlimited number of processors. In this case the test of Theorem 3.1 must be used to decide whether two tasks should execute simultaneously or not based on the amount of data that must be exchanged between them. A modification of the TACOM algorithm can be used in this case to compute the optimal partitioning and the optimal schedule when it is guaranteed that if needed, each task can be allocated as many processors as it requests. In what follows we use exactly the same notation as in the previous section.

As mentioned earlier r_i is the number of processors requested by task v_i of G. When we consider a composite task v_i^c that contains more than one task of G, we define r_i^s and r_i^f to be the number of processors requested by the first and last task-components of v_i^c respectively. The order is taken to be that which is implied by the direction of the data dependence arcs. For example, the composite task corresponding to node $v_9^c = (2, 5)$ in the graph of Figure 3.3, has $r_9^s = r_2$ and $r_9^f = r_5$. The execution times of the composite tasks of L are given by

$$t_{ij} = \sum_{m=i}^{j-1} t_{m,m+1} \tag{3.22}$$

where $t_{m,m+1}$ is given by (3.6). In this case (3.22) replaces (3.19) in the TACOM algorithm. The analogous modification of (3.20) for labeling the nodes of L is given by (3.23).

$$x_m = \min_r \left\{ \frac{|r_{h_r}^f - r_{mi}^s|}{\max(r_{h_r}, r_{mi}^s)} * w_{h_r} + \max(x_{h_r}, t_{mi}) \right\}. \tag{3.23}$$

Similarly the arcs of L are labeled in this case using

$$w_{e_2} = w_{e_1} + \frac{|r_{m,i}^f - r_{i+1,g}^s|}{\max(r_{m,i}^f, r_{i+1,g}^s)} * w_i \tag{3.24}$$

instead of (3.21) of the previous section. TACOM can then be applied in exactly the same way as shown in Figure 3.4. The output of the algorithm will be the optimal partition of a chain graph that minimizes communication for the unlimited processor case without reducing the amount of available parallelism.

3.6. Reducing Communication in Triangles

As mentioned earlier the problem of minimizing communication in general DAGs without reducing potential parallelism is NP-Complete. However we can solve the problem optimally for chains and other simple graphs that are often the building blocks of general DAGs. We can then

process DAGs by solving the communication problem optimally for each of their building blocks. Local optimality however does not guarantee global optimality, but this approach can be used to design near-optimal heuristics.

One of the most frequently observed basic subgraphs in general DAGs are triangles, that arise from transitive dependences. Figure 3.5 depicts a triangle with tasks a, b, and c. Weights w_1, w_2, and w_3 denote as usual the amount of communication between pairs of tasks. We can have an arbitrary number of tasks between nodes a and b for example, but since they would form a chain they can be partitioned using TACOM. Therefore in this case we are only interested in finding the conditions for the best merging of tasks in a triangle. Exhaustive search is appropriate in this case. There are five possibilities: To leave the triangle unchanged, to merge tasks a and b, or a and c, or b and c, or all a, b, and c. The best solution will be dictated based on the amount of data communication and the relative size of the tasks. This is trivial to determine. Let T_1^a, T_1^b, and T_1^c be the serial execution times of tasks a, b, and c respectively. Without loss in generality assume also that $T_1^a \geq T_1^b \geq T_1^c$. Consider the following enequalities.

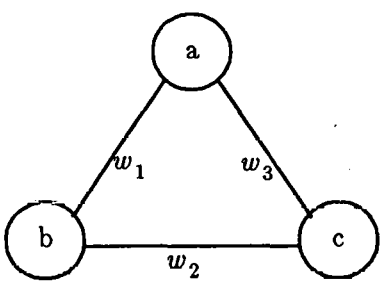


Figure 3.5. A basic subgraph.

$$T_1^b - T_1^c \leq w_1 - w_3 \quad \text{(X1)}$$

$$T_1^a - T_1^c \leq w_1 - w_2 \quad \text{(X2)}$$

$$T_1^b \leq w_1 - w_2 \quad \text{(X3)}$$

$$T_1^a - T_1^b \leq w_3 - w_2 \quad \text{(X4)}$$

$$T_1^c \leq w_3 - w_2 \quad \text{(X5)}$$

$$T_1^a \leq w_1 + w_3 \quad \text{(X6)}$$

$$T_1^b \leq w_1 + w_2 \quad \text{(X7)}$$

$$T_1^c \leq w_2 + w_3 \quad \text{(X8)}$$

then we have the following proposition.

Proposition 3.1 If the tasks of a triangle must be merged to reduce communication, then the best merging is specified in the following table.

Condition	Best Merging
$X6 \bullet X7 \bullet X8$	merge all a, b, and c
$(X1 \bullet X2) + (X1 \bullet X3)$	merge a and b
$(\overline{X1} \bullet X4) + (\overline{X1} \bullet X5)$	merge a and c
$(\overline{X2} \bullet \overline{X4}) + (\overline{X3} \bullet \overline{X5})$	merge b and c

where $\bullet$ and $+$ denote AND and OR operations respectively.

Proof For each case we compare the corresponding execution times. For example merging all three tasks together gives the best solution, if the resulting execution time is less than or equal to the execution time corresponding to merging any pair of tasks only, e.g. merging tasks a and b. In other words,

$$T_1^a + T_1^b + T_1^c \leq \max(T_1^a + T_1^b,\ T_1^c) + w_2 + w_3.$$

By simplifying this enequality we get (X6). Similarly we can derive (X7) and (X8). All other cases are similar. ■

The tests of the above proposition can be used to determine the optimal or near-optimal merging patterns for other basic subgraphs as well. As an example consider the case of four tasks with data interdependences that form a square. By adding a zero communication link between a pair of nonadjacent tasks, we can form two triangles that can now be processed separately using the previous proposition. Note that by definition independent tasks can never be merged.

3.7. Constructing the Task Graph of Fortran Programs

The assumption about constant communication weights used previously is not very realistic when we consider real Fortran programs. Communication per se, i.e., the number of data items that must be transmitted between two given tasks is indeed constant. However in multiprocessor systems the time it takes to transmit the same amount of data at two different instances may vary. In our case we want to measure the effect of communication on program speedup, that is, in terms of processor latencies, or execution time. Consider for example two tasks u and v, where v is data dependent on u. The number of data items that will be sent from u to v is constant for each such pair of tasks. In order to measure communication overhead precisely using a deterministic model, we must assume that the time it takes to transmit a unit of data between two processors is constant. Delays in processor initiation caused by communicating these data however vary and depend on several factors. One such factor is the relative position of the source of a dependence in the code of task u, and its sink in the code of task v.

Consider for example the parallel execution of tasks u and v of the previous paragraph, when both tasks start executing at the same moment each on a different processor. Let τ be the time it takes to transmit a unit of data between two processors. Assume that each statement in u and v takes a unit of time to execute. Consider now a data dependence from u to v caused by a variable X which is defined in u and used in v. Let d_1 be the execution time of the segment of

u between its first statement and the statement defining X, inclusive. Correspondingly d_2 denotes the execution time of the code of task v between its first statement and the statement using X, exclusive. Depending on the relative positions of the use and the definition statements of X in v and u respectively, we have the four cases described below and shown in Figure 3.6.

Case 1: X is computed in u before it is used in v. Figure 3.6a shows the case where $d_2 > d_1 + \tau$. The communication overhead in this case is zero, that is, no extra delays will occur in the

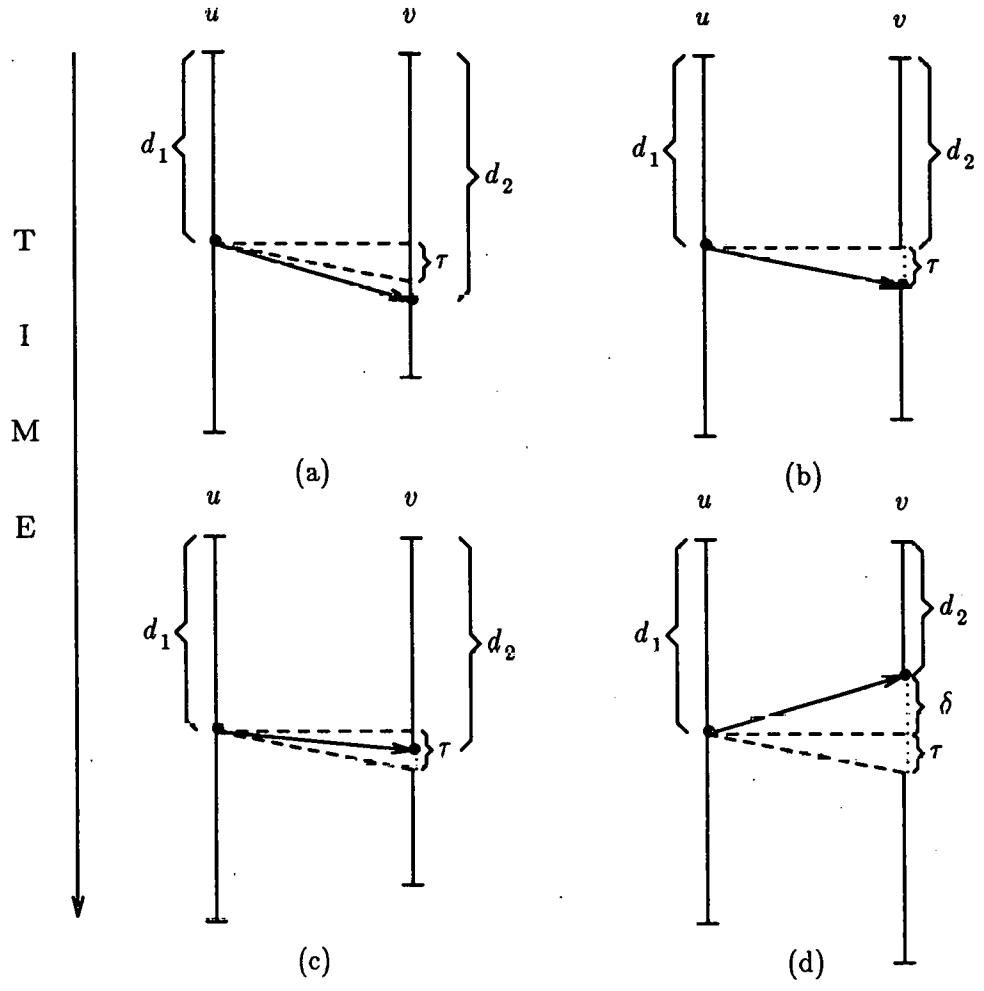


Figure 3.6. Four different cases of interprocessor communication.

processor executing v, since the value of X will be available when needed.

Case 2: Figure 3.6b shows the case where X is computed in u when it is needed in v. That is, $d_2 = d_1$ and in this case the communication overhead is τ, the time it takes to transmit X between the two processors.

Case 3: In Figure 3.6c X is used after it is computed but $d_2 - d_1 < \tau$, and the overhead in this case is $(d_1 - d_2) + \tau$.

Case 4: In this case (Figure 3.6d) X needs to be used before it is computed. This case involves the largest overhead, and the processor latency is given by $(d_1 - d_2) + \tau$.

It is clear that the communication overhead in all four cases is given by

$$\delta = \max(0,\ d_1 - d_2 + \tau),$$

and the execution time of u and v when they execute concurrently on two processors is given by

$$\max(T_1^u,\ T_1^v + \delta).$$

The two tasks u and v should therefore be merged if and only if

$$T_1^u + T_1^v \leq \max(T_1^u,\ T_1^v + \delta)$$

or equivalently if and only if $T_1^u \leq \delta$. The same analysis can be done when each of u and v execute on several processors, and both start executing concurrently. In this case we assume that each task is distributed equally among p processors, and each processor executes $\lceil T_1^u/p \rceil$ and $\lceil T_1^v/p \rceil$ part of u or v respectively. The corresponding timing in this case would use $\lceil d_1/p \rceil$ and $\lceil d_2/p \rceil$ in place of d_1 and d_2 respectively.

This model which is realistic satisfies Lemma 3.1, Theorem 3.1, and most of the assumptions used by the simple model employed in the previous sections. The TACOM algorithm is also valid here as well. However, intertask communication should be estimated by the compiler for

each different merging pattern that appears in the layered graph of Figure 3.3.

The data communication overhead caused by several dependences can be determined in a similar way. Consider again the tasks u and v of Figure 3.6. Only "parallel" dependences, that is dependences that lexically do not intersect need to be considered. This implicitly assumes that in the worst case we can transmit $\lceil \tau/\epsilon \rceil$ data items simultaneously through an interconnection network without conflicts (where ϵ is the execution time of a single statement). Clearly this is a realistic assumption for a parallel processor system. When two or more dependence arcs cross each other, the arc whose sink precedes all other (sinks) is preserved and all other dependences are ignored (as far as processor initiation delays are concerned, but are accounted for in the amount of communication). An example with two dependence arcs is shown in Figure 3.7a. A communication overhead may be caused by e_2 but not by e_1 and therefore e_1 is discarded as far as overhead is concernecd.

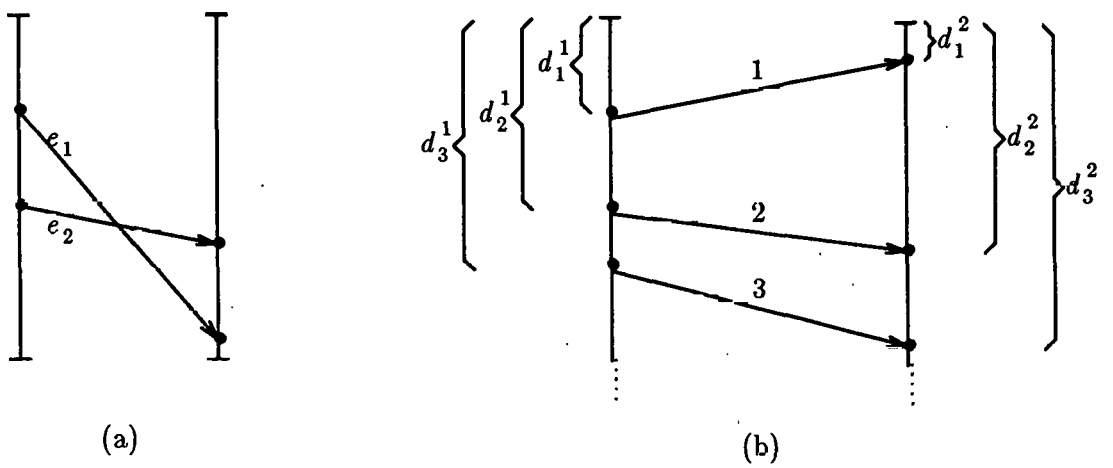


Figure 3.7. (a) A cross dependence example. (b) Parallel tasks with several dependences.

Thus in the general case we have nonintersecting dependences from u to v that may prolong the execution time of v as shown in Figure 3.7b. (Since strongly connected components are always merged, as discussed below, the case with dependences going both directions never arises.) If d_i^1 and d_i^2 denote the execution times for the segments of u and v defined by the i-th dependence arc, as shown in Figure 3.7b, then we have the following proposition.

Proposition 3.2 If there are n dependences from u to v and the two tasks execute concurrently on two different processors, the total communication overhead will be

$$\delta_n = \max\left(0, (d_n^1 - d_n^2) + \tau + \delta_{n-1}\right)$$

where $\delta_0 = 0$.

Proof The formula can be easily proved using induction on n. (For example, $\delta_1 = \max(0, d_1^1 - d_1^2 + \tau)$ is obviously true as it was shown above for the case of Figure 3.6.) ■

The execution time is then given by

$$\max(T_1^u, T_1^v + \delta_n)$$

and the two tasks are merged if and only if $T_1^u \leq \delta_n$.

We have seen how interprocessor communication can be measured for Fortran programs. Assuming that the compiler is used to measure interprocessor communication as described above, the task composition algorithm can be applied to find the optimal partitions for certain types of Fortran code. For long chunks of straight-line code, optimal partitions can be obtained by TACOM easily. The same is true for chains of serial loops, assuming loop bounds and branch probabilities of conditional statements are known. In the case where all loops have bounds that are expressions of the same variable, optimality can also be achieved.

Even though partitioning of Fortran programs can be performed using the bottom-up approach described earlier, we choose to partition programs using the following heuristic guide-

lines for reasons that are stated later.

Partitioning Heuristic

> **R1:** Outermost loops form a single task (that can possibly generate several processes during execution).
>
> **R2:** Conditional and unconditional branching statements form individual tasks (that are given the highest priority during execution).
>
> **R3:** Each one of the remaining statements form individual tasks.
>
> **R4:** Strongly connected components of tasks of type R3 are merged into single tasks.
>
> **R5:** The TACOM algorithm is applied to chains of serial loops, or chains of tasks of type R3 or R4.

Any chains in the resulting program graph cannot be further reduced. Next we look for tasks of type R3 or R4 with transitive interdependences and for trees of tasks of type R3 or R4. In the case of trees, the root-to-leaf path that involves the highest total communication is determined and treated as a chain. TACOM can then be applied to that chain adjusting the appropriate dependences during merging. The same procedure is repeated until the entire tree is reduced to a single task, or the root-to-leaf path of a reduced tree cannot be reduced further.

Theorem 3.1 and thus TACOM can be applied to only special cases of chains of parallel loops. These include cases of pairs of loops with interdependences such that all the dependences from each iteration of the first loop point to a single iteration of the second loop.

As mentioned above, outermost loops are considered to be special tasks, and no partitioning is attempted inside an outermost loop. There are two reasons for this approach: First interprocessor communication for parallel loops (usually) has a "regular" pattern, and therefore is easier to deal with. Secondly, we have developed algorithms (discussed in following chapters) that process complex loops efficiently or optimally in a separate way.

If the number of processors p, that will be used during execution is known in advance, program partitioning can be carried out further so that the resulting task graph G has enough "parallelism" and "balanced" tasks that involve very little communication. As explained earlier there is no universal definition of "parallelism" or "degree of parallelism" but it intuitively means that G should have enough tasks to assign to all p processors at any moment during execution. Since G is a directed acyclic graph it can be transformed into a layered graph as explained in Chapter 2. If the i-th layer has n_i tasks and there are k layers in total, a possible definition of the *degree of parallelism* δ_π (useful only to deterministic approaches) can be

$$\delta_\pi = \frac{1}{k} \sum_{i=1}^{k} n_i .$$

In summary, the techniques of this chapter can be used to obtain the partitioning of a Fortran program into tasks that are parallel or serial outermost loops, or scalar tasks. The resulting partitioning involves minimum communication between a series of scalar tasks and a series of serial loops, for any scheduling algorithm. This partitioning in effect defines a program task graph that can be scheduled on a parallel machine using techniques that are described in later chapters.

CHAPTER 4

OPTIMAL LIMITED PROCESSOR ALLOCATION TO PARALLEL LOOPS

4.1. Optimal Processor Assignment to Parallel Loops

It has been shown that in most programs parallel loops are the source of the greatest percentage of parallelism [Kuck84]. In this chapter we will investigate the problem of processor assignment to parallel loops. This problem becomes especially important when we deal with nested parallel loops where inefficient assignment algorithms may result in an execution time far worse than the optimal. In programs with several nested parallel loops the efficiency may then drop down to unacceptable levels. We can informally define the limited processor assignment problem as follows: Given an arbitrary multiply nested loop which contains serial and parallel (DOACR, DOALL) loops and a number of P processors, find the best way of assigning the P processors to the loops so that the parallel execution time of the entire module is minimized. For loops with very few nest levels and systems with a small number of processors exhaustive search might be affordable at compile time. But as the number of processors increases, the number of processor-loop combinations grows exponentially. Moreover, loops with large nest levels are not very uncommon in scientific computations. As an example, 10 to 17 deeply nested parallel loops were observed in several subroutines of the restructured (by Parafrase) IEEE Digital Signal Processing Package [IEEE79]. In this chapter we define the static processor allocation problem for parallel loops, and discuss an optimal algorithm based on dynamic programming that handles loops of arbitrary complexity. The algorithm described here may also be used to determine locally optimal assignments of loops of different programs when throughput is to be maximized in a multiprogrammed parallel processor system.

4.1.1. Optimal Simple Processor Assignment to DOALLs

A metric called the *efficiency index* is used throughout this chapter. The usefulness of this metric is twofold. First it makes it easier to formulate the processor assignment problem, and secondly it allows us to observe several interesting properties of the problem that are otherwise hidden in modular arithmetic.

A processor assignment algorithm (OPTAL) is proposed that solves the general problem optimally. The optimal processor assignment is guided by the use of a function called the *assignment function*. The assignment function can easily be defined to measure efficiency or parallel execution time. Processor assignment in an arbitrarily nested parallel loop is performed by allocating (possibly) different numbers of processors to different loops in the nest. The techniques described below partition a p-processor machine hierarchically and assign different partitions of processors to different loops in the nest. To simplify our terminology we invariably refer to partitions of any size as processors. Consider for example two nested DOALL loops N_1 and N_2 and a 6 processor system. One possible allocation to these loops would be one that assigns 2 processors to the outer loop and 3 processors to the inner loop. More precisely, this means that the machine is partitioned into two halfs, each half consisting of 3 physical processors. Then each iteration of the outer loop N_1 is assigned a one half partition and the corresponding iterations of the inner loop are allocated 3 physical processors. Therefore, "processor" is used as a generic term in this chapter and refers to clusters of physical processors of different sizes.

Before we discuss processor assignment issues we need to introduce some notation and definitions. To simplify the notation, each loop is assumed to be normalized (i.e., its iteration space is of the form 1,...,N) and denoted by the upper bound of its iteration space. Thus, N_i denotes a DO whose loop body is executed N_i times, and $L=(N_1, N_2, \ldots, N_m)$ denotes an m level nested DO where N_i is surrounded by N_{i-1} and surrounds N_{i+1}, $(i=2, \ldots, m-1)$. N_1

and N_m are the outermost and innermost loops respectively.

In what follows the number of available processors P is always assumed to be "useful", that is, less than or equal to the maximum number of processors that a loop L can fully utilize. Here we assume that a fixed number of processors has been allocated to each outermost loop of a given program. Our aim is to optimally distribute the allocated processors to the different loops in the (arbitrarily complex) nest for each such outermost loop, in order to minimize the parallel execution time. How processors are allocated to different outermost loops of a given program is the subject of Chapter 6.

Definition 4.1. For a DOALL with N_i iterations that has been assigned p_i processors we define ϵ_i, the *efficiency index* or *EI* of N_i as follows:

$$\epsilon_i^{p_i} = \frac{N_i / p_i}{\lceil N_i / p_i \rceil}. \tag{4.1}$$

The efficiency index is an indicator of how efficiently a loop runs on a given number of processors. The higher the EI the higher the efficiency (as defined in Chapter 1). Some other properties of the efficiency index that will be used directly or indirectly in the following sections are:

P1: For any DOALL N_i and any number of processors p we have: $0 < \epsilon_i^p \leq 1$.

P2: For any N_i, $\epsilon_i^1 = 1$.

P3: For $N_i \geq p$, $\epsilon_i^p > 1/2$.

It should also be noted that $p \neq q$ does not necessarily imply $\epsilon_i^p \neq \epsilon_i^q$. It is always true that $p_i \geq 1$. If during allocation a loop N_i is not assigned processors explicitly, it is implied that $p_i = 1$ and thus $\epsilon_i = 1$.

Definition 4.2: For a nested DOALL $L = (N_1, N_2, \ldots, N_m)$, a number of processors $P = p_1 p_2 \ldots p_m$ and a particular assignment of P to L we define the *efficiency index vector*

$\omega = (\epsilon_1^{p_1}, \epsilon_2^{p_2}, \ldots, \epsilon_m^{p_m})$ of L, where $\epsilon_i^{p_i}$ is the EI for loop N_i using p_i processors.

In what follows, the terms "assignment of P" and "decomposition of P" are used interchangeably. Any assignment of P to L defines implicitly a decomposition of P into factors $P = p_1 p_2 \ldots p_m$ where each of the m different loops receives p_i, $(i = 1,2,\ldots,m)$ processors. A processor assignment profile $(p_1, p_2, \ldots, p_m)$ (where loop N_i receives p_i processors) can also be described by its efficiency index vector as defined above.

Definition 4.3: For an assignment $\omega = (\epsilon_1^{p_1}, \ldots, \epsilon_m^{p_m})$ of $P = p_1 p_2 \ldots p_m$ processors to L, we define E_L, the *compound efficiency index* (*CEI*) of L as

$$E_L = \prod_{i=1}^{m} \epsilon_i^{p_i}. \tag{4.2}$$

For any L we also have $0 < E_L \leq 1$. Let T_1 be the serial execution time of a perfectly nested DOALL L. Next suppose that L is executed on P processors and let T_P and T_P' denote the parallel execution times for two different assignments $\omega = (\epsilon_1, \epsilon_2, \ldots, \epsilon_m)$ and $\omega' = (\epsilon_1', \epsilon_2', \ldots, \epsilon_m')$ of P to L, where $P = p_1 \ldots p_m = p_1' \ldots p_m'$. We can express the parallel execution time T_P of L in terms of its CEI as follows:

$$T_P = \prod_{i=1}^{m} \lceil N_i / p_i \rceil B, \quad \text{or} \quad \frac{1}{T_P} = (1/B) \frac{\prod_{i=1}^{m} N_i / p_i}{\prod_{i=1}^{m} \lceil N_i / p_i \rceil} \prod_{i=1}^{m} p_i / N_i =$$

$$(1/B) \left(\prod_{i=1}^{m} \frac{N_i / p_i}{\lceil N_i / p_i \rceil} \right) \frac{\prod_{i=1}^{m} p_i}{\prod_{i=1}^{m} N_i} = \prod_{i=1}^{m} \epsilon_i^{p_i} \frac{P}{NB} = \frac{E_L P}{NB} \quad \text{or}$$

$$T_P = \frac{NB}{\mathrm{E}_L P} \tag{4.3}$$

where B is the execution time of one iteration of the loop, and $N = \prod_{i=1}^{m} N_i$. B is assumed to be a constant for all loop iterations. The following lemma is a direct application of (4.3).

Lemma 4.1: $T_P < T_P^{'}$ if and only if $\mathrm{E}_L > \mathrm{E}_L^{'}$.

In the next few sections we show how the efficiency index can be used to direct the efficient assignment of processors to perfectly nested parallel loops.

Given a nested loop L and a number of processors P we call a *simple* processor assignment one that assigns all P processors to a single loop N_i of L. A *complex* processor assignment is one that assigns two or more factors of P to two or more loops of L.

Theorem 4.1. The optimal simple processor assignment over all simple assignments of P to L, is achieved by assigning P to the loop with $\epsilon = \max_{i=1,m} \{\epsilon_i^P\}$.

Proof: Without loss in generality assume that $\epsilon = \epsilon_1$. Then $\epsilon_i = 1$ for $i = 2,3,\ldots,m$ and $E_L = \epsilon_1$. For any other simple allocation of P to N_j, $j \neq 1$, with a CEI of $E_L^{'}$, we have $\epsilon_1 = E_L \geq E_L^{'} = \epsilon_j$. The optimality follows from Lemma 4.1.■

The following two lemmas are indirectly used in subsequent discussion.

Lemma 4.2. $\prod_{i=1}^{m} \left\lceil \frac{N_i}{p_i} \right\rceil \geq \left\lceil \prod_{i=1}^{m} \frac{N_i}{p_i} \right\rceil$.

Proof: By definition we have that

$$\prod_{i=1}^{m} \left\lceil \frac{N_i}{p_i} \right\rceil \geq \prod_{i=1}^{m} \frac{N_i}{p_i} \tag{4.4}$$

and since the left hand side of (4.4) is an integer it follows directly that

$$\prod_{i=1}^{m} \left\lceil \frac{N_i}{p_i} \right\rceil \geq \left\lceil \prod_{i=1}^{m} \frac{N_i}{p_i} \right\rceil. \blacksquare$$

The next lemma follows directly from Lemma 4.2.

Lemma 4.3. For any integer n, we have $n\left\lceil \frac{N}{pn} \right\rceil \geq \left\lceil \frac{N}{p} \right\rceil$.

For the next lemma and most of what follows we assume that processors are assigned in units that are equal to products of the prime factors of P unless explicitly stated otherwise. Therefore each loop is assigned a divisor of P including one.

Lemma 4.4. If N is a (single) DOALL loop, $P = p_1 p_2 \ldots p_m$, and ϵ_i, $(i=1,2,\ldots,m)$ are the efficiency indeces for assigning p_1, p_1p_2, $p_1p_2p_3$, ..., $p_1p_2 \ldots p_m$ processors to N respectively, then

$$\epsilon_1 \geq \epsilon_2 \geq \epsilon_3 > \ldots \geq \epsilon_m. \tag{4.5}$$

Proof: We sketch the proof for $m=2$ and the general case follows exactly the same reasoning.

$$\epsilon_1 \geq \epsilon_2 \rightarrow \frac{N / p_1}{\lceil N/p_1 \rceil} \geq \frac{N/p_1p_2}{\lceil N / p_1p_2 \rceil} \rightarrow p_2 \left\lceil \frac{N}{p_1p_2} \right\rceil \geq \left\lceil \frac{N}{p_1} \right\rceil$$

But the last relation is Lemma 4.3 and is therefore true. ■

From Lemma 4.1 we conclude that the optimal processor assignment of P to L is the one that maximizes E_L. Each assignment defines indirectly a decomposition of P into a number of factors less than or equal to the number of loops in L. As P grows, the number of different decompositions of P into factors grows very rapidly. From number theory we know that each integer is

uniquely represented as a product of prime factors. Theorem 4.2 below can be used to prune (eliminate from consideration) several decompositions of P, or equivalently several assignment profiles of P to L that are not close to optimal. From several hand generated tests we observed that the use of Theorem 4.2 in a branch and bound algorithm for determining the optimal assignment of processors eliminated more than 90% percent of all possible assignments. In some instances all but the optimal assignments were pruned by the test of Theorem 4.2.

Again, let $L = (N_1, \ldots, N_m)$ be a perfectly nested DOALL that executes on P processors, and $P = p_1 p_2 \ldots p_k$ be any decomposition of P where $k \leq m$. Now let $\epsilon = \max_{1 \leq i \leq m} \{\epsilon_i^P\}$ be the maximum efficiency index over all simple assignments of P to L, and $\epsilon_i = \max_{1 \leq j \leq m} \{\epsilon_j^{p_i}\}$, $(i=1, 2,\ldots, k)$ be the maximum efficiency indeces (over all loops of L) for the factors $p_1, p_2, \ldots, p_k$ of P respectively (i.e., $\epsilon_j^{p_i} = (N_j/p_i)/(\lceil N_j / p_i \rceil)$). Note that here we do not perform any actual assignment of processors to loops, but simply compute the maximum efficiency index for each factor p_i of P over all loops of L excluding the loop that corresponds to ϵ. If T_s and T_c are the parallel execution times for L corresponding to the optimal simple assignment of P, and the optimal complex assignment of the specific factors of P respectively, and S_s and S_c their respective speedups, we have the following theorem (using the notation of this paragraph).

Theorem 4.2. If there exists $i \in \{1,2,\ldots,k\}$ for which $\epsilon \geq \epsilon_i$, then $T_s \leq T_c$ and thus $S_s \geq S_c$. Or equivalently if one of the factors of P has a maximum efficiency index equal to or less than the maximum efficiency index of P, then we gain more speedup by assigning the entire P to a single loop than from any complex assignment of the factors of P (including the optimal).

Proof: Without loss of generality we can assume that the optimal complex allocation assigns

```
      DOALL 1 I1=1,63
        DOALL 2 I2=1,7
          DOALL 3 I3=1,31
            DOALL 4 I4=1,20
                    .  .  .
                    .  .  .
                    .  .  .
4           CONTINUE
3         CONTINUE
2       CONTINUE
1     CONTINUE
```

If P=32 the optimal assignment is that which assigns all 32 processors to the outermost loop.

Figure 4.1. An application of Theorem 4.2.

more than one processor to the first k loops ($k \leq m$), and implicitly one processor to the remaining $m - k$ loops. Therefore the corresponding efficiency index vector for the optimal complex allocation is $\omega_c = (\epsilon_1^{p_1}, \ldots, \epsilon_k^{p_k}, 1,\ldots, 1)$ and for the optimal simple allocation of P is $\omega_s = (1,\ldots, 1, \epsilon, 1,\ldots, 1)$, where ϵ corresponds to the j-th position. Then the parallel execution times of the optimal simple and complex allocations are:

$$T_s = N_1 N_2 \ldots N_{j-1} \left\lceil \frac{N_j}{P} \right\rceil N_{j+1} \ldots N_m \quad \text{and} \quad T_c = \left\lceil \frac{N_1}{p_1} \right\rceil \ldots \left\lceil \frac{N_k}{p_k} \right\rceil N_{k+1} \ldots N_m \tag{4.6}$$

Suppose now that for some $i \in \{1,2,\ldots,k\}$ we have $\epsilon \geq \epsilon_i$ or equivalently,

$$\frac{N_j/P}{\lceil N_j / P \rceil} \geq \frac{N_i / p_i}{\lceil N_i / p_i \rceil} \rightarrow \frac{N_j}{p_1 \ldots p_i \ldots p_k} \left\lceil \frac{N_i}{p_i} \right\rceil \geq \frac{N_i}{p_i} \left\lceil \frac{N_j}{P} \right\rceil \tag{4.7}$$

Again without loss in generality we may assume $j > i$ and by multiplying both sides of (4.7) by $N_1 \ldots N_{i-1} N_{i+1} \ldots N_{j-1} N_{j+1} \ldots N_m$ we have

$$\frac{N_1}{p_1}\cdots\frac{N_{i-1}}{p_{i-1}}\left\lceil\frac{N_i}{p_i}\right\rceil\frac{N_{i+1}}{p_{i+1}}\cdots\frac{N_k}{p_k}\cdots N_m \geq N_1\cdots N_{j-1}\left\lceil\frac{N_j}{P}\right\rceil N_{j+1}\cdots N_m. \qquad (4.8)$$

If we denote the left and right hand sides of (4.8) by M1 and M2 respectively, we have $T_c \geq \text{M1} \geq \text{M2} = T_s$ or finally,

$$T_c \geq T_s \quad \text{and} \quad \text{S}_c \leq S_s. \qquad \blacksquare$$

Thus, given any decomposition of P into factors $P = p_1\cdots p_k$, a necessary (but not sufficient) condition for a complex assignment to be better than the best simple assignment is $\epsilon < \epsilon_i$, for all $i=1,2,\ldots,k$ (where ϵ_i is the maximum efficiency index for factor p_i over all loops of L). Obviously if $\epsilon = 1$ the optimal simple assignment is the overall optimal as well. An example of the application of Theorem 4.2 is shown in Figure 4.1. The next theorem is a generalization of Theorem 4.2 when only factors of P are considered.

Theorem 4.3. If in Theorem 4.2 $P=p_1p_2\cdots p_k$ is the prime factor decomposition of P and there exists $i\in\{1,\ldots,k\}$ for which $\epsilon \geq \epsilon_i$, then the simple allocation of P to L (corresponding to ϵ) is the overall optimal.

Proof: Suppose that for some $i\in\{1,\ldots,k\}$ we have $\epsilon \geq \epsilon_i$. Suppose further that ϵ_i corresponds to loop N_i. We therefore have that

$$\epsilon_i \geq \epsilon_j \quad \text{for all} \quad \text{N}_\text{i} \neq N_j \quad \text{for the same} \quad \text{p}_\text{i}. \qquad (4.9)$$

Any other allocation of P to L defines another decomposition $P = p_1'p_2'\cdots p_r'$, $r \leq k$ and since p_i is a prime factor of P there exists some p_i' that includes p_i (that is, $p_i' = p_ip_i''$, for some integer p_i''). We consider the following two scenarios:

Case 1: Suppose that in this new allocation p_i' is assigned to the same loop N_i as p_i in the prime factor allocation. Then if ϵ_i' is the efficiency index of p_i' and since $p_i' = p_ip_i''$ it follows directly from Lemma 4.4 that $\epsilon_i \geq \epsilon_i'$ and therefore $\epsilon \geq \epsilon_i \geq \epsilon_i'$.

Case 2: Suppose now that $p_i^{'}$ is assigned to a loop different than N_i, say N_j. If ϵ_j and $\epsilon_j^{'}$ are the efficiency indeces for allocating p_i and $p_i^{'}$ processors to N_j respectively, then from Lemma 4.4, the initial hypothesis, and relation (4.9) we have

$$\epsilon \geq \epsilon_i \geq \epsilon_j \geq \epsilon_j^{'}. \tag{4.10}$$

Therefore in any complex allocation of P to L, there is at least one loop of L that has a maximum efficiency index less than or equal to ϵ, the efficiency index of the optimal simple allocation (OSA) of P to L. Thus the OSA is the overall optimal. ■

Corollary 4.1. If $N = \prod_{i=1}^{m} N_i$, ϵ is the efficiency index of the optimal simple assignment, ϵ_i is the efficiency index for the i-th loop in a complex optimal assignment ($i=1, 2,..., m$), and E_L the corresponding compound efficiency index, then any optimal complex assignment must satisfy,

$$\epsilon < \epsilon_i \leq 1, \quad (i=1, 2,..., m) \quad \text{and} \quad \epsilon < E_L \leq 1.$$

Let

$$E_0 = \frac{N / P}{\lceil N/P \rceil} \quad \text{where} \quad N = \prod_{i=1}^{m} N_i.$$

Then any optimal assignment of P to L satisfies

$$E_L \leq E_0 \tag{4.11}$$

where E_L is the CEI of an optimal assignment. Only in special cases would there be an optimal assignment of P to L for which the equality in (4.11) holds. A compiler transformation called loop coalescing, that is discussed in the next chapter, can be applied to certain types of loops and always achieves $E_L = E_0$.

Corollary 4.1 can be used to check whether a given complex assignment is better than an optimal simple assignment. It would be useful however to be able to answer the question of the existence of such assignments. That is, given a loop L and a number of P processors, is there an

optimal complex assignment better than the optimal simple assignment? If for a particular loop the answer is negative, the optimal simple assignment is chosen and therefore the problem for that loop is solved in constant time, assuming the efficiency indeces have been computed. Proposition 4.1 below provides the test for the existence of an optimal complex assignment.

For each loop $N_i \in L$, we define the *critical capacity* g_i of N_i as the maximum number of processors that can be assigned to N_i with its efficiency index remaining strictly greater than ϵ (the maximum efficiency index of P). In other words, for each N_i, g_i is chosen to satisfy,

$$\epsilon_i^{g_i} > \epsilon \quad \text{and} \quad \epsilon_i^{g_i+r} \leq \epsilon$$

for any $r \geq 1$. Then we have the following proposition.

Proposition 4.1. A necessary condition for the existence of a complex assignment of P to L which is better than the corresponding optimal simple assignment, is

$$\prod_{i=1}^{m} g_i \geq P.$$

Proof: If we had $\prod_{i=1}^{m} g_i < P$, then in any complex allocation there would be at least one loop N_i with $\epsilon_i \leq \epsilon$ (assuming that all P processors are useful). From Theorem 4.2 then it follows that the optimal simple allocation is also the overall optimal.■

The obvious approach for optimally solving the general instance of the static processor assignment problem is exhaustive search. For small nested loops and a very small number of processors exhaustive search would probably be tolerable at compile time. For medium size loops and a few tens of processors however, the cost of exhaustive search becomes intolerable even at compile time. For example the number of different assignments of 50 processors to 15 nested loops is 4.8×10^{13}. If it takes 1000ns (on a fast machine) to process each different assignment it would

take more than 555 days CPU time to find the optimal assignment of 50 processors to 15 loops.

Using the results of this section however, we can design a branch and bound algorithm that greatly reduces the number of candidate optimal assignments. In several cases the tests of this section can prune all possible assignments but the optimal and in practice such a branch and bound algorithm would have polynomial complexity for most cases. The problem remains unsolved though since we can never guarantee polynomial complexity and we can always come up with an example loop which can make even the branch and bound algorithm run in exponential time.

In the next section we present an optimal processor assignment algorithm that has a low polynomial complexity and finds the optimal assignment for all types of loops and any number of processors.

4.1.2. Optimal Complex Processor Assignment to Parallel Loops

In order to better illustrate the ideas of this section we start by considering perfectly nested DOALLs and a number of $P=2^k$ processors. As we proceed the concepts are generalized to include more complex loop structures such as nonperfectly nested combinations of serial, DOALL, and DOACR loops.

Let us consider an m-level nested DOALL $L=(N_1, N_2, ..., N_m)$ and a number of $P=2^k$ processors. The *optimal allocation* algorithm or OPTAL which is analytically described below will give us the optimal assignment profile of the P processors to the m loops of L. For each loop L we compute the *efficiency table* M as shown in Figure 4.2. Each column j of M corresponds to a loop N_j of L and each row i corresponds to a number of 2^i, $(i=0,1,...,k)$ processors. An entry (i,j) of table M contains the efficiency index for assigning 2^i processors to loop N_j. This $(m \times k)$ efficiency table will be used repeatedly by OPTAL to obtain the optimal

assignment of P to L.

From Lemma 4.4 we observe that each column of M is ordered in nonincreasing order. If the loops are ordered by size then each row of M is also ordered in nonincreasing order. Therefore if ϵ_{ij} is the element of M in the i-th row and j-th column,

$$\epsilon_{ij} \geq \epsilon_{iw} \quad \text{for} \quad w \geq j.$$

It is clear that in any assignment of P to L there can be at most one entry of the lower half of M involved in that assignment. Let us give an outline of the basic steps of the algorithm. The process starts by assigning the P processors to the innermost or outermost loop, and let us always start from the innermost in our case. The second step finds the optimal assignment of P to the two innermost loops. In the process we also need to compute the optimal assignment of $1, 2, 2^2, \ldots, 2^k$ processors respectively to the two innermost loops. These assignments however are computed only once for each loop and stored for later use by successive steps.

In general, after the $(m - i)$-th step OPTAL has found the optimal assignment of $1, 2, 2^2, \ldots, P$ processors to loops $L_i=(N_i, N_{i+1}, \ldots, N_m)$. The next $(m - i + 1)$-th step

	N_1	N_2	N_3	.	.	.	N_m
1	ϵ_{11}	ϵ_{12}	ϵ_{13}	.	.	.	ϵ_{1m}
2	ϵ_{21}	ϵ_{22}	ϵ_{23}	.	.	.	ϵ_{2m}
2^2	ϵ_{31}	ϵ_{32}	ϵ_{33}	.	.	.	ϵ_{3m}
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
2^k	ϵ_{k1}	ϵ_{k2}	ϵ_{k3}	.	.	.	ϵ_{km}

Figure 4.2. The efficiency table for $P=2^k$ and m nested loops.

considers loop N_{i-1} and finds the optimal assignment of 1, 2, 2^2, . . . , P processors to loop (N_{i-1}, L_i), possibly by reassigning processors from L_i to N_{i-1}. All possible assignments for N_{i-1} are considered. Note that all possible assignments for L_i have already been computed. At the end of the m-th step OPTAL outputs the profile of the optimal assignment of $P = 2^k$ to loop L=(N_1, N_2, ..., N_m). Based on Lemma 4.1 the optimal assignment of P to L would be the one that maximizes E_L. This is precisely what OPTAL does.

4.1.2.1. The Perfectly-Nested Loop Case

In this section we describe in detail processor assignment for perfectly nested DOALLs and $P = 2^k$. We use this case as an example of the application of the general algorithm which is described in the next section. It is followed by a simple example that illustrates the details of computing the optimal assignment. The heart of OPTAL is a recursive function G, that is defined as follows: Given P=2^k and L an m-way nested loop as previously, we define $G_i(q)$ as the product of efficiency indeces of the optimal assignment of q processors to loops (N_i, N_{i+1}, ..., N_m). More specifically a closed form expression of function $G_i(q)$ is given by,

$$G_i(q) = \max_{1 < p_j < q} \left\{ \prod_{j=i}^{m} \epsilon_j^{p_j} \right\}$$

and such that $q = \prod_{j=i}^{m} p_j \leq P$. The recursive definition of $G_i(q)$ and the one that we will be using from now on is given by (4.12)

$$G_i(P) = \max_{0 \leq r \leq k} \left\{ \epsilon_i^{2^r} G_{i+1}(P/2^r) \right\} \quad \text{or} \tag{4.12}$$

$$G_i(P) = \max \left\{ G_{i+1}(P),\ \epsilon_i^2 G_{i+1}(P/2),\ \epsilon_i^4 G_{i+1}(P/4),\ \epsilon_i^8 G_{i+1}(P/8),\ \ldots,\ \epsilon_i^P G_{i+1}(1) \right\}$$

where ϵ_i^q is the efficiency index for assigning q processors to loop N_i (available from table M). Relation (4.12) tells us that the optimal assignment of P to (N_i, N_{i+1}, ..., N_m) can be found by selecting from all assignments of 2^r processors to loop N_i and 2^{k-r} processors to (N_{i+1}, ..., N_m), ($r = 0, 1, ..., k$), the one that maximizes $\prod_{j=i}^{m} \epsilon_j$.

The function in (4.12) is computed for $i=m, m-1, ..., 1$ and for each i we also compute $G_i(1)$, $G_i(2)$, $G_i(2^2)$,..., $G_i(P=2^k)$. The optimal assignment of P to L will be given at the end of the m-th step by $G_1(2^k)$. Initially (first step) for $i=m$ we have $G_m(q) = \epsilon_m^q$. For each $G_i(q)$ the corresponding processor assignment profile is stored and when $G_1(2^k)$ is computed the profile for the optimal assignment is available.

The algorithm completes in m steps. In each of the m steps, $k = \log_2 P$ function evaluations are performed and each of the ($r=1,2,...,k$) function evaluations involves the computation of the maximum of r values. The overall complexity of the algorithm is therefore $O(m \log^2_2 P)$. Using the results of the previous sections, we can easily avoid unnecessary computations and further reduce the complexity of OPTAL.

The explicit processor assignment vector (with the exact number of processors assigned to each loop) is computed as a side effect of the computation of G_i. When a particular G_i is chosen as optimal, the corresponding assignment vector can be trivially reconstructed. In order to illustrate the computational details of the algorithm let us consider a simple example involving four DOALLs and 2^5 processors. It should be noted that this approach not only finds the optimal assignment of the given P processors to a particular loop nest, but it also finds the optimal assignments of $P/2$, $P/4$, $P/8$, $P/16$,..., 1 processors to the same loop. We can therefore determine the minimum number of useful processors with little extra cost.

Example 4.1.1: Consider the loop $L=(N_1 = 15, N_2 = 17, N_3 = 17, N_4=25)$ of Figure 4.3

```
      DOALL 1 I1=1,15
        DOALL 2 I2=1,17
          DOALL 3 I3=1,17
            DOALL 4 I4=1,25
                  .   .   .
                  .   .   .
                  .   .   .
4               CONTINUE
3             CONTINUE
2           CONTINUE
1         CONTINUE
```

Figure 4.3. The nested loop of example 4.1.1.

	1	2	4	8	16	32
$G_4()$	1	$\frac{25}{26}$	$\frac{25}{28}$	$\frac{25}{32}$	$\frac{25}{32}$	$\frac{25}{32}$
$G_3()$	1	$\frac{25}{26}$	$\frac{25}{26}*\frac{17}{18}$	$\frac{25}{28}*\frac{17}{18}$	$\frac{25}{32}$	$\frac{25}{32}$
$G_2()$	1	$\frac{25}{26}$	$\frac{25}{26}*\frac{17}{18}$	$\frac{25}{26}*(\frac{17}{18})^2$	$\frac{25}{28}*(\frac{17}{18})^2$	$\frac{25}{32}$
$G_1()$	1	$\frac{25}{26}$	$\frac{15}{16}$	$\frac{15}{16}$	$\frac{15}{16}$	$\frac{15}{16}*\frac{25}{26}$

Figure 4.4. The table for example 4.1.1.

and let $P=2^5$. The optimal assignment of P to L is computed as follows: First the 5×4 efficiency matrix M is computed. At the first step for $i=4$ we have $G_4(2^r) = \epsilon_4^{2^r}$ for $r = 0, 1,..., 5$. The computations for the remaining three steps are shown analytically below. In each case the

maximum element appears in bold letters. For each of the four steps the optimal assignments are tabulated in table T shown in Figure 4.4. Each row of T corresponds to each of the four steps.

Step 2

$$G_3(2) = \max\left\{ \mathbf{G}_4(2),\ \epsilon_3^2 \mathrm{G}_4(1) \right\}$$

$$G_3(4) = \max\left\{ G_4(4),\ \epsilon_3^2 \mathbf{G}_4(2),\ \epsilon_3^4 G_4(1) \right\}$$

$$G_3(8) = \max\left\{ G_4(8),\ \epsilon_3^2 \mathbf{G}_4(4),\ \epsilon_3^4 G_4(2),\ \epsilon_3^8 G_4(1) \right\}$$

$$G_3(16) = \max\left\{ \mathbf{G}_4(16),\ \epsilon_3^2 \mathrm{G}_4(8),\ \epsilon_3^4 \mathrm{G}_4(4),\ \epsilon_3^8 \mathrm{G}_4(2),\ \epsilon_3^{16} \mathrm{G}_4(1) \right\}$$

$$G_3(32) = \max\left\{ \mathbf{G}_4(32),\ \epsilon_3^2 G_4(16),\ \epsilon_3^4 G_4(8),\ \epsilon_3^8 G_4(4),\ \epsilon_3^{16} G_4(2),\ \epsilon_3^{32} G_4(1) \right\}$$

Step 3

$$G_2(2) = \max\left\{ \mathbf{G}_3(2),\ \epsilon_2^2 G_3(1) \right\}$$

$$G_2(4) = \max\left\{ \mathbf{G}_3(4),\ \epsilon_2^2 G_3(2),\ \epsilon_2^4 G_3(1) \right\}$$

$$G_3(8) = \max\left\{ G_3(8),\ \epsilon_2^2 \mathbf{G}_3(4),\ \epsilon_2^4 G_3(2),\ \epsilon_2^8 G_3(1) \right\}$$

$$G_2(16) = \max\left\{ G_3(16),\ \epsilon_2^2 \mathbf{G}_3(8),\ \epsilon_2^4 G_3(4),\ \epsilon_2^8 G_3(2),\ \epsilon_2^{16} G_3(1) \right\}$$

$$G_2(32) = \max\left\{ \mathbf{G}_3(32),\ \epsilon_2^2 G_3(16),\ \epsilon_2^4 G_3(8),\ \epsilon_2^8 G_3(4),\ \epsilon_2^{16} G_3(2),\ \epsilon_2^{32} G_3(1) \right\}$$

Step 4

$$G_1(2) = \max\left\{ \mathbf{G}_2(2),\ \epsilon_1^2 G_2(1) \right\}$$

$$G_1(4) = \max\left\{ G_2(4),\ \epsilon_1^2 G_2(2),\ \epsilon_1^4 \mathbf{G}_2(1) \right\}$$

$$G_1(8) = \max\left\{ G_2(8),\ \epsilon_1^2 G_2(4),\ \epsilon_1^4 G_2(2),\ \epsilon_1^8 \mathbf{G}_2(1) \right\}$$

$$G_1(16) = \max\left\{ G_2(16),\ \epsilon_1^2 G_2(8),\ \epsilon_1^4 G_2(4),\ \epsilon_1^8 G_2(2),\ \epsilon_1^{16} \mathbf{G}_2(1) \right\}$$

$$G_1(32) = \max\left\{ G_2(32),\ \epsilon_1^2 G_2(16),\ \epsilon_1^4 G_2(8),\ \epsilon_1^8 G_2(4),\ \epsilon_1^{16} \mathbf{G}_2(2),\ \epsilon_1^{32} G_2(1) \right\}$$

The optimal assignment in this example is therefore the one that assigns 16 processors to loop N_1 and 2 processors to loop N_4. The processor assignment profile is reconstructed as follows. First we look at the maximum element of $G_1(32)$. This element is $\epsilon^{16} G_2(2)$ which indicates that loop N_1 receives 16 processors, and the remaining processors are allocated to $G_2(2)$. The maximum element of entry $G_2(2)$ in Step 3 is $G_3(2)$ which indicates that loop N_2 receives 1 processor. Continuing in the same way, the maximum element of entry $G_3(2)$ is $G_4(2)$ which again indicates that loop N_3 is assigned 1 processor, and therefore loop N_4 is assigned the remaining 2 processors.

4.1.2.2. The General Algorithm

Although most real polyprocessor systems have $P = 2^k$ for some integer k, OPTAL can be used to generate optimal processor assignments for any integer P. It also handles arbitrarily nested parallel loops. Before we describe the details of the general algorithm however, we need to define the concepts of DOACR and loop nesting more precisely.

As mentioned in Chapter 1, a DOACR is a parallel loop in which data dependences allow for partial overlap of successive iterations during execution on an MES system. In other words, if iteration i starts at time t on a given processor, iteration $(i+1)$ can start at time $t + d$, where d is (idealy) a constant. Constant d is called delay and represents the execution time of a subset of loop statements whose data dependence graph forms a cycle. If B is the (serial) execution time of the loop body, then d/B gives the percentage of overlap, (or doacross percentage). When $d=B$ the loop is serial while if $d=0$ the loop is a DOALL. DOALL and serial loops are therefore special cases of DOACR loops. The parallel execution time of a DOACR loop with N_i iterations, a delay of d_i and a body size of B_i that executes on P processors is given by the following [PoBa86].

$$T_P^i(B_i) = \left(\left\lceil \frac{N_i}{P} \right\rceil - 1\right) * \max\{B_i, Pd_i\} + d_i * \Big((N - 1) \bmod P\Big) + B_i \qquad (4.13)$$

In order to simplify the notation in the following discussion, we assume that a block of assignment statements (BAS) can be considered as a DOACR loop with $N_i = 1$, and $d_i = 0$.

An arbitrarily complex nested loop can be uniquely represented as a k- level tree where k is the maximum nest depth. The leaves of the tree correspond to BASs and intermediate nodes correspond to (DOACR) loops. The total number of nodes in a loop tree is $\lambda + \mu$, where λ is the number of individual loops in the structure and μ the number of BASs. An example of a nested loop and its tree representation are shown in Figure 4.5. Intermediate tree nodes at level m correspond to loops at nest depth m. We assume that individual loops in an arbitrarily nested loop are numbered increasingly in lexicographic order (Figure 4.5).

In the general case loops are not perfectly nested and therefore the efficiency index as defined in Section 4 is not useful. We can redefine the efficiency index for the general case but it is more convenient to define the assignment function to measure directly parallel execution time.

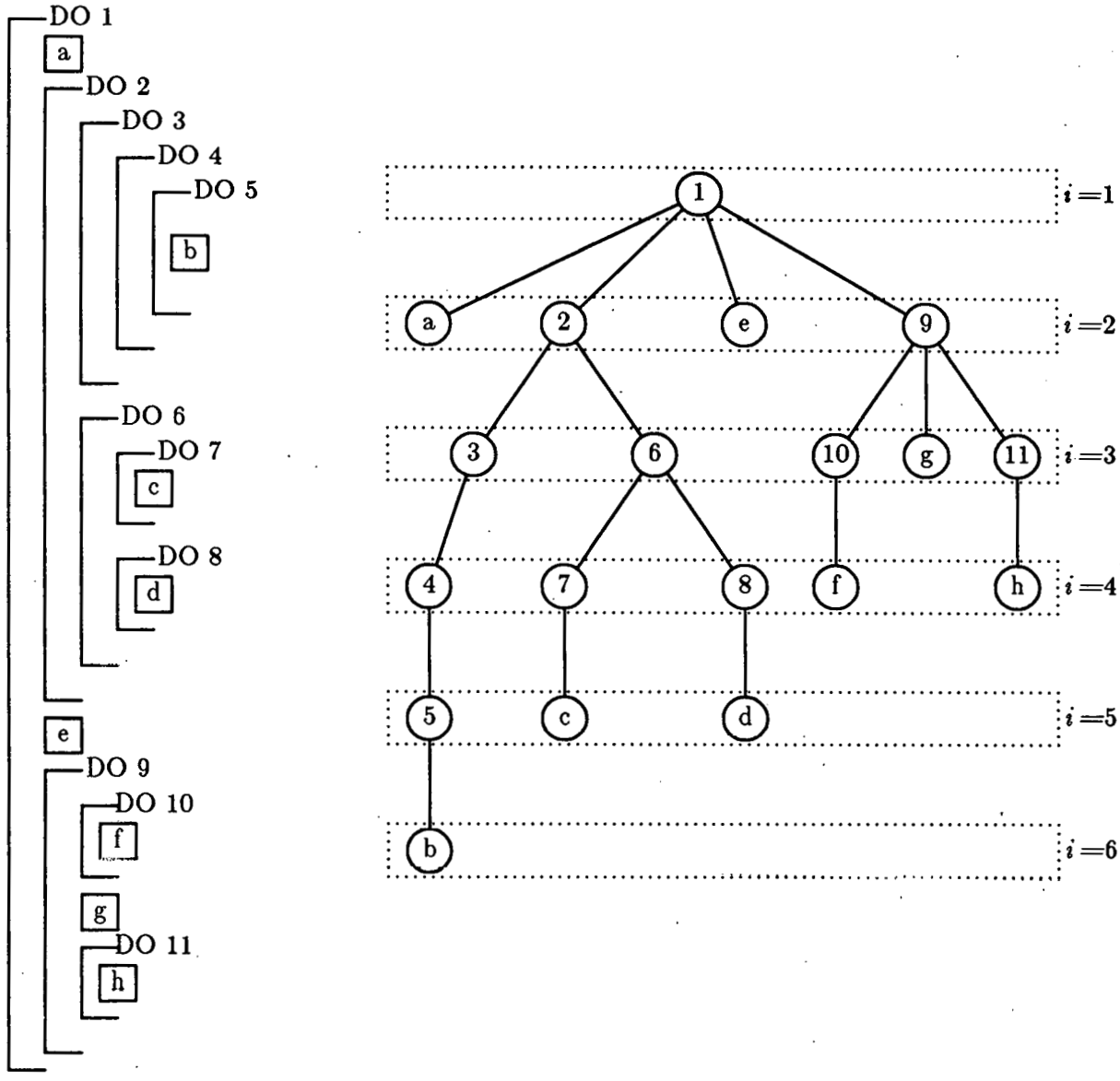


Figure 4.5. A nested loop and its tree representation. Squares and leaves denote BASs.

The max term of the assignment function in the previous section becomes min in this case since our objective is to minimize execution time and thus maximize speedup.

The steps of the general algorithm are almost identical to the case of perfectly nested loops. The example of Figure 4.5 is used whenever it helps illustrate the computations involved. A $\lambda \times P$ table can be used to store intermediate values. (λ, P are the numbers of loops and processors respectively.) During the first step we compute the parallel execution time of the DOACR loops at level k on the tree, where k is again the maximum nest level. This is done as follows:

$$G_k^i(q) = T_q^i(B_i), \quad (q = 1, 2, ..., P) \qquad (4.14)$$

and for all leaves i.

where T_q^i is given by (4.13). The general step is defined recursively as in the perfectly nested loop case. The optimal assignment of P processors to loops in levels i through k ($i < k$), (assuming the optimal assignment of P to loops at level $i+1$ is known), is then generated by:

$$G_i^j(q) = \min_{1 \le r \le q} \left\{ T_r^j \left(\sum_{n \text{ child of } j} G_{i+1}^n(\lfloor q/r \rfloor) \right) \right\} \qquad (4.15)$$

and for (q=1, 2, 3, ..., P)

where (4.15) is computed for all nodes (loops) j at level i, and $T(*)$ is given by (4.13). The summation in (4.15) accounts for all nodes at level $i+1$ that are descendants of node j, that is, all loops nested inside loop N_j. The optimal assignment of P processors to a given loop is given by $G_1^1(P)$. Recall from the example of the previous section that the detailed processor assignment vector is automatically constructed during the evaluation of (4.15). For each loop the number of processors assigned to it corresponds to the minimum term in (4.15). It should be noted that all optimal assignments of 1, 2,..., P-1 processors to L are computed as intermediate results of the computation of $G_1^L(P)$. We therefore have the following.

Lemma 4.5 The maximum number of useful processors given P for a loop L is the minimum Q, such that $1 \le Q \le P$ and $G_1^L(Q)$=$G_1^L(P)$.

OPTAL

Input:
A loop $L=(N_1,\ N_2,\ \ldots,\ N_m)$, of nest depth nest k and P processors.

Output:
An optimal processor allocation profile of P to L.

Method:

- 1. For all loops j at level k of the loop tree, and
  For all $q = 1, 2, 3, \ldots, P$, compute the allocation function:
  $G_k^j(q) = T_q^j(B_j)$

- 2. For $i=(k-1)$ to 1 Compute:
  For all loops j in nest depth i compute:

$$G_i^j(1) = T_1^j\left[\sum_{n\ child\ of\ j} G_j^n(1)\right]$$

$$G_i^j(2) = \min\left\{\left[T_1^i\left[\sum_{n\ child\ of\ j} G_{i+1}^n(2)\right]\right],\quad \left[T_2^i\left[\sum_{n\ child\ of\ j} G_{i+1}^n(1)\right]\right]\right\}$$

$$\vdots$$

$$G_i^j(P) = \min\left\{\left[T_r^j\left[\sum_{n\ child\ of\ j} G_{i+1}^n(\lfloor P/r \rfloor)\right]\right],\ \text{for } r=1, 2, 3, \ldots, P\right\}$$

  and store the results in the j-th row of table T.
- 3. Output the processor allocation profile corresponding to $G_1(P)$.

Figure 4.6. The processor allocation (OPTAL) algorithm.

A procedural description of OPTAL is given in Figure 4.6.

Theorem 4.4 For any loop L of maximum nest depth m, and any integer P, OPTAL terminates after m iterations and generates the optimal assignment of P processors to L.

Proof: The proof is by induction on i. Since i is decreasing in successive steps, we apply induction backwards. For $i=m$ (the innermost loop) we have by definition an optimal allocation given

by $G_m(P)$. Suppose that for $i=k+1$, $G^j_{k+1}(q)$, $(q=1, 2, ..., P)$ is optimal, for all loops j at nest level $k+1$. We will show that for $i=k$, $G^j_k(q)$, $(q=1, 2,..., P)$ is also optimal. For every q, G_k is defined by (4.15) and without loss of generality we can assume that

$$G^j_k(q) = T^j_r\left[\sum_{n \text{ child of } j} G^n_{k+1}(\lfloor q/r \rfloor)\right]. \tag{4.16}$$

Since $G^n_{k+1}(\lfloor q/r \rfloor)$ is optimal for all j by the induction hypothesis, and since (4.16) is the minimum term in (4.15), it follows that $G^j_k(q)$ is optimal for q and thus $G^j_k(P)$ is optimal. We thus conclude that $G^1_1(P)$ gives the optimal allocation of P processors to loop L. ■

The complexity of the algorithm can be easily determined. The assignment function G^j_i is computed P times for each node (loop) in the tree, or a total of λP times. Each evaluation of the assignment function also involves finding the minimum of an average of $P/2$ terms. The complexity therefore (without counting additions) is $O(\lambda P^2 / 2)$. The complexity can be reduced to $O(\lambda P \log P)$, and OPTAL can be used to implement a systolic array control unit that consists of $P \log P$ nodes and determines the optimal assignment of P processors to a given loop in λ steps, as discussed later in this chapter.

Note that the (maximum) speedup resulting from the optimal assignment of P processors to a loop L is given by,

$$S_P = \frac{G^L_1(1)}{G^L_1(P)}.$$

An interesting point of this approach is that although loops at the same nest level are allocated the same total number of processors, each loop manages (assigns) its own processors to its own iterations in an independent way. For example, suppose that loops 3 and 6 of Figure 4.5 are allocated 8 processors each. A possible assignment then may assign 1 processor to loops 3 and 4,

Subroutine Name	S_{32}	S_{256}	S_{2048}
ELMBAK	31.9	242.0	668.0
ELMHES	31.7	33.6	33.6
ELTRAN	29.3	71.3	84.5
HQR2	18.0	26.0	28.0
TRED1	31.0	235.0	240.0
MINFIT	28.0	130.0	181.0
TRED2	18.3	36.5	39.5
CBABK2	30.0	53.5	57.4
CH	25.0	66.9	85.0
COMBAK	31.9	248.5	721.0
CORTB	32.0	254.0	1250.0
CORTH	32.0	252.0	501.0
BANDV	31.0	98.0	98.0

Table 4.1: Speedup values for 32, 256, and 2048 processors for EISPACK subroutines.

Subroutine Name	S_{32}	S_{256}	S_{2048}
INISHL	32.0	255.8	1021.0
WFTA	32.0	255.8	2036.9
TRBIZE	30.8	128.6	128.6
PCORP	31.9	246.3	537.0
POWER	26.4	68.2	79.6
COSYFP	22.2	54.4	65.7
FREDIC	31.9	162.0	190.0
FLPWL	30.9	169.4	363.0
DIINIT	28.0	89.5	119.4
SRINIT	21.3	48.6	56.8
SMINVD	31.9	120.6	186.0
DEFIN4	19.2	37.6	37.6
FFT	30.8	191.0	505.0
LOAD	22.0	36.3	36.3
COVAR1	31.5	68.0	76.5
CLHARM	27.7	91.8	120.0
FLCHAR	31.0	188.3	458.8
REMEZ	10.0	12.1	12.3
D	31.3	210.0	670.0
LPTRN	11.0	13.9	14.8

Table 4.2: Speedup values for 32, 256, and 2048 processors for IEEE DSP subroutines.

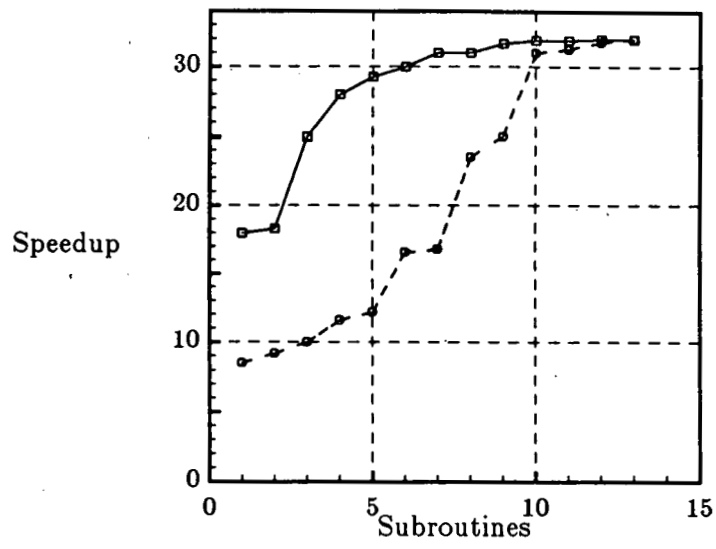
and 8 processors to loop 5, while in the second case we may have 2 processors assigned to loop 6, and 4 processors to each of the loops 7 and 8. It is clear that loops on the same nest level must be assigned the same total number of processors when executing on a parallel processor system. Otherwise we have suboptimal parallel execution times since some processors will be forced to remain idle.

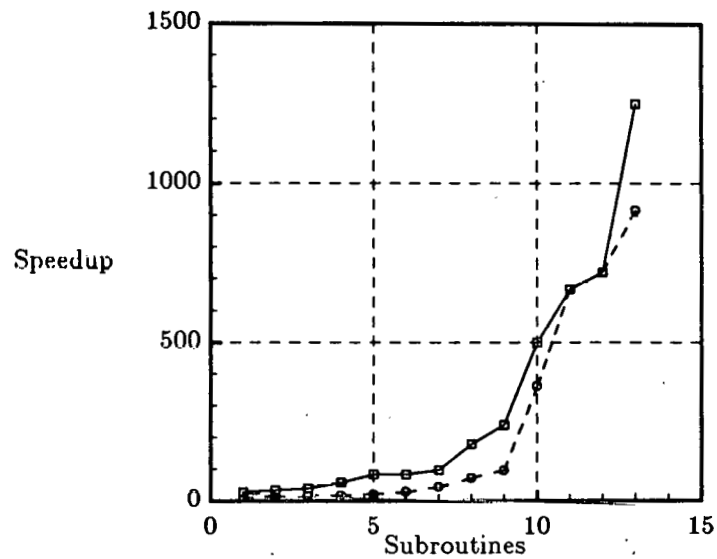
4.2. Experiments

We implemented this processor assignment algorithm in the Parafrase compiler. Processor assignment is performed after DOALL and DOACR loops are recognized and delays computed. In our experiments we measured speedup values for some subroutines of the EISPACK and IEEE DSP packages.

In our case T_P, the parallel execution time, was measured for P=32, P=256, and P=2048 processors, and for loop bounds set to 10. In some EISPACK subroutines where loop bounds correspond to the bandwidth of band-matrices, we used loop bounds of 1 or 4. The speedup values measured for the three different numbers of processors are shown in Tables 4.1 and 4.2. The subroutines from the two packages used in these experiments were randomly selected.

From the speedup values we observe that for 32 processors the average speedup is almost linear for both EISPACK and IEEE subroutines. For 256 processors the average speedup for EISPACK subroutines is about 137, or more than $P/2$. That is, we have an efficiency of more than 50% for P=256. For the IEEE subroutines we observe an even higher average efficiency for the same number of processors. The third column in each table corresponds to an unlimited number of processors. Since most of the EISPACK subroutines deal with square matrices, for 40 $\times$ 40 arrays the maximum expected speedup is 1600. Taking into account several loops with





Figures 4.7 and 4.8. New and previous speedups for EISPACK for 32 and 2048 processors.

bounds of 1 or 4 and the number of one-dimensional loops, the average maximum speedup should

be expected to be considerably lower than 1600. The average speedup of the third column of Table 4.1 is about 310, which corresponds to an average efficiency of about 15%. Since at most 1600 processors would be useful for most of the EISPACK routines, in reallity we would have an efficiency of about 20%. The corresponding values for the third column of Table 4.2 are quite higher than those of EISPACK. Generally, supercomputers deliver a wide range of performances from program to program. This is true of real machines [DoHi85], and has been observed in our earlier experimental work [Kuck84]. It appears, from the experiments we have conducted so far, that when OPTAL is used there is very little variation when programs are run with limited number of processors.

Considering the fact that efficiencies in the range of 20% are characterized very satisfactory in modern supercomputers, we can claim that optimal processor assignments to parallel loops result in high speedups for most cases. Processor allocations to independent code segments can increase the average speedup at least by a factor of two [Veid85].

Figures 4.7, 4.8, 4.9, and 4.10 show the improvement in speedup for the same set of EISPACK and IEEE/DSP subroutines. The horizontal axis in the plots correspond to subroutines arranged in order of increasing speedup. The vertical axis display actual speedups. The solid lines plot the speedup spectrum obtained by using OPTAL. The dotted lines plot speedups obtained by the previous method [Cytr84].

We compared the performance improvement using the two methods for $p=32$ and $p=2048$. As mentioned above, the problem size was chosen so that $p=2048$ approaches the unlimited processor case. We observe that for $p=32$ the speedup improvement is very significant for both EISPACK and IEEE/DSP routines. This is the case when the number of processors is small relative to the problem size, nonoptimal allocations have a significant negative impact on performance. In other words when the number of processors is relatively small we can not afford

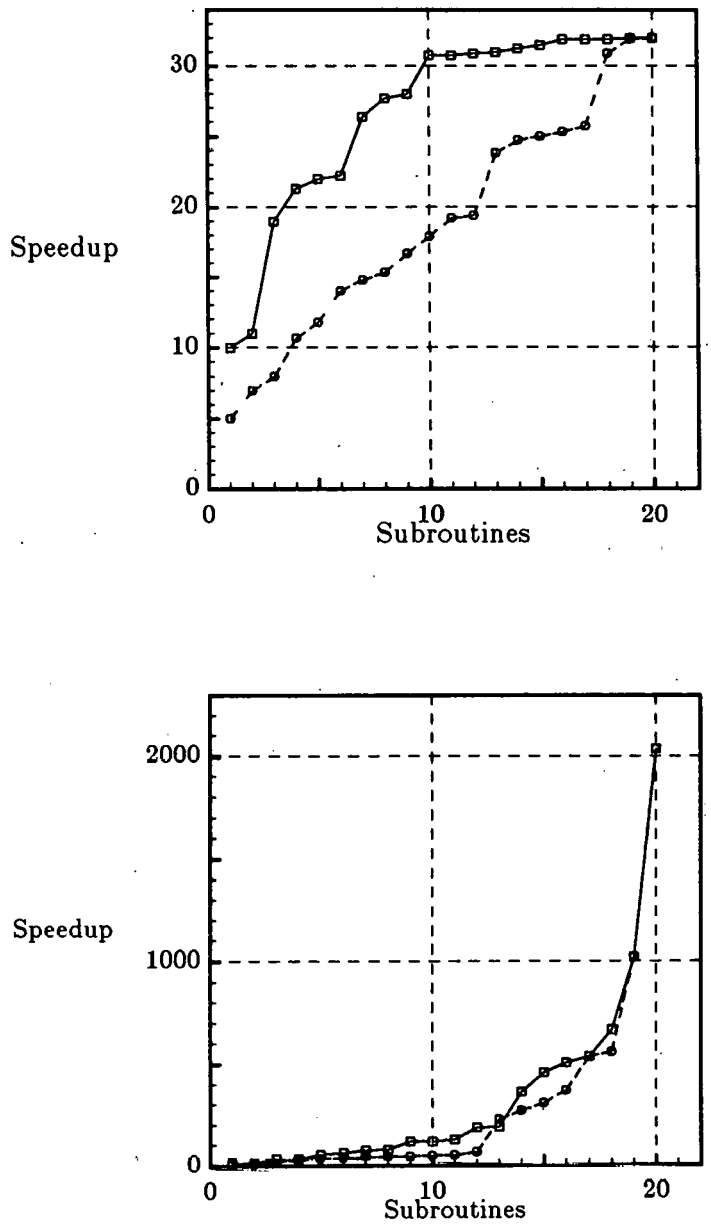


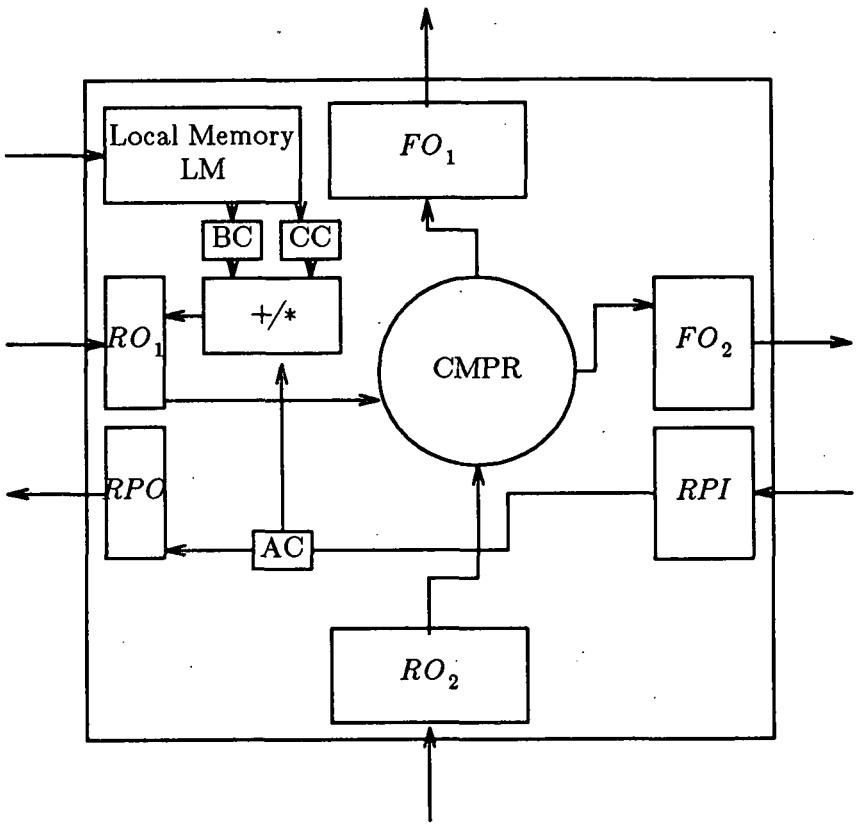
Figure 4.9 and 4.10. New and previous speedups for IEEE/DSP for 32 and 2048 processors.

to underutilize even a few processors. When the number of processors is large, the inefficiency

introduced by poor utilization of a few processors is amortized and has less impact on overall performance. This becomes evident in the plots of Figures 4.8 and 4.10 where the number of processors is 2048. In this case the improvement in performance is significantly less than in the case of $p=32$ (Figures 4.7 and 4.9). The relative performance improvement for $p=256$ lies in between.

4.3. Implementing OPTAL with Systolic Array

As mentioned earlier OPTAL generates optimal static processor assignments if the loop bounds are known at compile-time. This is frequently the case in numerical software where loop bounds usually reflect the problem size. However there are many cases where the loop bounds are not known at compile-time and default values are used by Parafrase instead. In such cases it is impossible to assure optimality. For example, loops with unknown loop bounds at compile-time are triangular loops whose bounds are actually indeces of outer loops. By unrolling loops that surround triangular loops we obtain a sequence of loops with constant upper bounds that can be handled optimally. This unrolling does not need to lexically take place but processor assignment can be performed assuming an implicit loop unrolling at compile-time. Loop upper bounds that cannot be estimated at compile-time are also those that are determined by a function call or by the value of an array element, for example. This problem is alleviated at run-time however, where loop bounds must be known before the loop is entered. It would thus be appropriate in such cases to perform processor assignment at run-time, just before we start executing each loop. Of course more information is available at run-time but the overhead of run-time assignment would also be more significant. Although run-time assignment and scheduling is the subject of Chapter 7, in this section we discuss a hardware implementation of OPTAL in the form of a systolic array that can be used for run-time processor assignment. Let us again consider the perfectly nested loop case where the number of processors is a power of two. The



AC, *BC*, *CC*: Registers that hold operands for the functional unit.
CMPR: Comparator, finds the *MAX* of two reals.
FO_1, FO_2: Output latches.
RO_1, RO_2: Input latches.
RPI, *RPO*: Input and output latches for propagating data.

Figure 4.11. The structure of the systolic array cell.

general case is also discussed.

Consider the quadraply nested loop in Figure 4.3 and the computation of the allocation function G in Example 4.1.1. Since the computation of the different steps involves computing the max of a set of elements, the algorithm is naturally offered for a parallel hardware

implementation that uses a tree-structure to fan-in the partial max terms. By looking at Example 4.1.1 we observe that the main operations are an initial multiplication and subsequent comparisons. We can easily implement OPTAL using a systolic array that is a triangle of cells of the type shown in Figure 4.12.

The basic cell, shown in Figure 4.11, has a small local memory *LM* whose size should be at least m words, (where m is the nest depth of the loop under consideration). A functional unit that is used to multiply efficiency indeces of two loops at the beginning of each phase, where a phase consists of all the computation involved for each nest level. *CMPR* is a comparator and the remaining elements are latches used to receive and forward partial results. Going back to Example 4.1.1 it is easy to see how the computation is performed within each cell of the systolic array. For simplicity let us assume that each multiplication takes one full clock cycle and each of the other operations take half a clock cycle to complete (low and high).

The systolic array for P=16 is shown in Figure 4.12. Cells in the array are numbered (i,j) where i is the row number and j is the position of that cell in the i-th row counting from left to right. The initialization of each phase involves a multiplication which is performed as follows.

$[CC \leftarrow LM(x)]$ (innermost loop only)
$BC \leftarrow LM(y)$
$RO_1 \leftarrow BC * AC$

where x and y are local memory addresses. In general, at the beginning of the k-th phase cell (i,j) will execute:

$$BC \leftarrow \epsilon_{m-k+2}^{2^{i-j+1}}.$$

For the innermost loop the computation of the efficiency indeces is done separately and the appropriate values are broadcast to the corresponding cells. For example, cell (i,j) will receive $\epsilon_m^{2^{j-1}}$. After the multiplication step for a given phase has been completed, the following two steps

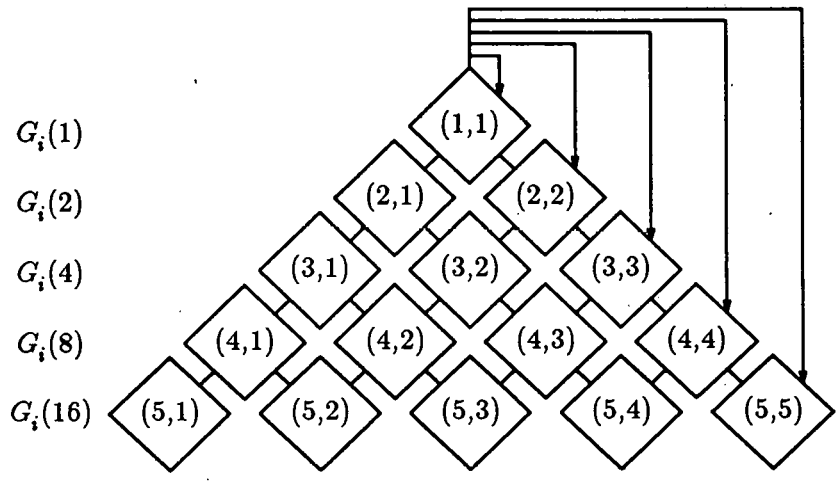


Figure 4.12. The systolic array implementation for $P=16$.

are repeated $\log P$ times.

Step 1 (High clock)

$$FO_1(i,j),\ FO_2(i,j) \leftarrow \max\left(RO_1(i,j),\ RO_2(i,j)\right).$$

Step 2 (Low clock)

$$RO_1(i,j) \leftarrow FO_1(i+1,j)$$
$$RO_2(i,j) \leftarrow FO_2(i,j+1)$$
$$RPI(i,j) \leftarrow RPO(i-1,j).$$

In Step 1 the operands in registers RO_1 and RO_2 within each cell are read by the *COMPR* and the result (max element) is latched into registers FO_1 and FO_2 of each cell. During the second step, each cell in the systolic array forwards the contents of its FO_1 and FO_2 registers to its two neighboring cells. Simultaneously, the contents of the *RPO* register within each cell are forwarded to the *RPI* and *AC* registers of their lower left neighbors.

As shown in Figure 4.12 each row i of cells in the array holds the elements of the set corresponding to $G_k(2^{i-1})$, $(k=m, m-1,..., 1)$. For the k-th loop, the computation of $G_k(P=2^n)$ is completed after $\log P$ steps. The intermediate allocation functions $G_k(1)$, $G_k(2)$, $G_k(2^2),...,G_k(2^{n-1})$ are also produced, one per clock, during that phase. Each result $G_k(2^r)$ is written into the AC registers of all cells numbered $(*, r+1)$. Clearly it takes $O(mlogP)$ steps to complete the optimal assignment of P processors to m nested loops.

Since the elements of the efficiency table M must be read into the cells of the systolic array (one row per cell), the computation time for M, which is mP would dominate the complexity of the entire computation. This problem however can be eliminated by overlapping the computation of M with the operation of the systolic array. Since each phase of computation in the array requires only one row of M, each row of M but the first, can be computed (and broadcast to the corresponding cells) during the computation of the previous phase. A linear systolic array can be used to implement OPTAL in a similar way.

The general case is similar but now we have to compute a time table instead of the efficiency table M. Each entry of the time table gives the execution time for the corresponding loop for a specific number of processors. Synchronization of the operations in the systolic array is an additional concern here. The number of clock cycles needed to complete one step varies with the number of additions needed in (4.15). Therefore, in order to synchronize the computation some of the cells may need to idle for a few clocks. As shown in (4.15) the number of additions in each phase depends on the number of loops at the same nest level.

CHAPTER 5

SCHEDULING WITH LOOP COALESCING

In the previous chapter we discussed compile-time scheduling, and presented an optimal deterministic processor assignment algorithm for arbitrarily complex parallel loops. Compile-time scheduling is a simple problem when we deal with singly nested loops where all loop iterations have equal execution times. In that case, the obvious one-step processor assignment is also the optimal one: the optimal distribution of N iterations to P processors is clearly the one that assigns $\lceil N/P \rceil$ iterations to each processor. It would be therefore desirable to have, if possible, parallel programs with singly nested parallel loops.

In this section we introduce a compiler transformation called *loop coalescing* that restructures certain types of multiply nested loops into single parallel loops. Thus, for those loops that can be restructured, the optimal processor assignment problem becomes simple. In addition, the processor assignments for the transformed loops are generally better than the optimal assignments to the original loops generated by OPTAL. This is true assuming all iterations of a loop have equal execution times. When this last condition is not satisfied the optimal processor assignment becomes a complex problem even for singly nested loops. In Chapter 7 we show how loop coalescing can be used to achieve optimal or near-optimal dynamic schedules for general parallel loops. This transformation is also used in Chapter 7 to reduce (and minimize in certain cases) the number of synchronization points needed during the execution of hybrid loops. Again we start from the perfectly nested loop case and generalize the concepts and results as we proceed. Some more definitions are in order.

Let $\underline{L} = (N_m, N_{m-1}, \ldots, N_1)$ be a perfectly nested DOALL and P the number of available processors. Let $N = \prod_{i=1}^{m} N_i$, B is the execution time of the innermost loop body, and

$$T_P^o = \left\lceil \frac{N}{P} \right\rceil B \tag{5.1}$$

where T_P^o represents the minimum execution time of loop L on P processors. Consider now any allocation ω of the P processors to the component loops of L, and let ω_0 be the optimal such allocation. If T_P^{ω} and $T_P^{\omega_0}$ denote the parallel execution time of L for the allocations ω and ω_0 respectively, then

```
      DOALL 1 I=1,15
        DOALL 2 J=1,7
              .  .  .
              .  .  .
              .  .  .
2       ENDOALL
1     ENDOALL
```

For P=27 the optimal processor allocation (OPTAL) assigns 3 processors to outer loop and 7 processors to inner loop which results in 5 iterations. The corresponding superoptimal allocation assigns 27 processors to 105 iterations which results in a total of only 4 parallel iterations.

Figure 5.1. Scheduling with coalescing.

```
DO 1 I=1,N
 DO 2 J=1,M                                  DO 1 J=1,N*M

    A(I,J)=B(I,J)          →                    A(J,1)=B(J,1)

2  ENDOALL                                  1 ENDOALL
1 ENDOALL
```

Figure 5.2. Example of loop collapsing.

$$T_P^{\omega} = \prod_{i=1}^{m} \left\lceil \frac{N_i}{q_i} \right\rceil B$$

for some distribution of q_i, $(i = 1, 2,..., m)$ processors to the component-loops N_i, $(i = 1, 2,..., m)$ of L such that $\prod_{i=1}^{m} q_i \leq P$. From Lemma 4.2 it then follows that

$$T_P^{o} \leq T_P^{\omega_o} \leq T_P^{\omega}.$$

Definition 5.1 An allocation ω of P processors to a multiply nested loop L is said to be *superoptimal* if and only if

$$T_P^{\omega} = T_P^{o}. \tag{5.2}$$

Obviously (5.2) holds true for all singly nested loops, but in general, is not true for multiply nested loops. It becomes evident therefore that transforming arbitrarily complex loops into single loops, not only simplifies the processor assignment problem, but it also improves the resulting

---

```
DOALL 1 J=1,N
  DOALL 2 K=1,N

    A(J,K)= ....

2   ENDOALL
1 ENDOALL
```

becomes

```
DOALL 1 I=1,N**2

    A(⌈I/N⌉ , I - N⌊(I - 1)/N⌋) = ....

1 ENDOALL
```

Figure 5.3. Loop coalescing in two dimensions.

---

schedules.

By applying loop coalescing we can achieve superoptimal limited allocations for the majority of DOALL loops. Loop coalescing transforms a series of nested DOALLs to a single DOALL with an iteration space equal to the product of the iteration spaces of the original loop. Then the superoptimal allocation is accomplished in a single step by allocating all P processors to the transformed loop. In order to apply loop coalescing to a nest of DOALLs, all dependence directions must be "=" [Wolf82]. In Chapter 7 we show how loop coalescing can be used with unequal direction vectors. Consider for example the loop of Figure 5.1 that is to be executed on a P=27 processor system. The optimal deterministic assignment to the original loop allocates 3 (clusters of) processors to the outer loop and 7 processors to the inner loop. This results in a total of 5 iterations per processor. If the original loop is coalesced into a single DOALL with 105 (=15*7) iterations, all processors are assigned to that single loop which results in 4 iterations per processor.

Loop coalescing resembles loop collapsing, another transformation that already exists in Parafrase. Loop collapsing though is different than coalescing in both its purpose and mechanism. The former is a memory related transformation that collapses doubly nested loops only, to single loops by transforming two dimensional arrays into vectors. Figure 5.2 shows an example of loop collapsing. The purpose of this transformation is to create long vectors for efficient execution on memory-to-memory SFA systems (e.g., CDC Cyber 205). No subscript manipulation is attempted by loop collapsing, which by the way, is applicable only to double perfectly nested DOALLs.

Loop coalescing should be applied so that the original and the transformed loops are semantically equivalent. This means that the transformation should manipulate loop subscripts so that there always exists a one-to-one mapping between the array subscripts of the original and

J	K	I
1	1	1
1	2	2
1	3	3
.	.	.
1	N	N
2	1	N+1
2	2	N+2
.	.	.
2	N	2N
.	.	.
.	.	.
.	.	.
N	1	(N-1)N+1
N	2	(N-1)N+2
.	.	.
N	N	NN

Figure 5.4. Index values for original and coalesced loop - two dimensions.

the transformed loop. Moreover, the resulting loop should be scheduled such that each processor knows exactly which iterations of the original loop it has been assigned. Since the resulting loop has a single index, we must find mappings that correctly map subscript expressions of the original loop (which are multivariable integer functions) to expressions involving a single subscript (corresponding to the index of the restructured loop).

Before we describe the general transformation let us look at two examples of loop coalescing. Figures 5.3 and 5.5 show the cases of coalescing perfectly nested DOALLs of nest depth two and three. Consider first the loop of Figure 5.3 and its coalesced equivalent. Figure 5.4 shows the index values for the two cases in the order they are assumed. Clearly the first subscript J of $A(J,K)$ should be transformed into an expression involving I, i.e.,

```
      DOALL 1 J=1,N
        DOALL 2 K=1,N
          DOALL 3 L=1,N

            A(J,K,L)= ....

3         ENDOALL
2       ENDOALL
1     ENDOALL
```

becomes

```
      DOALL 1 I=1,N^3

        A(⌈I/N^2⌉ , ⌈I/N⌉ - N⌊(I-1)/N^2⌋, I - N⌊(I - 1)/N⌋)=....

1     ENDOALL
```

Figure 5.5. Loop coalescing in three dimensions.

$$J \rightarrow f(I)$$

where f is an integer-value function and such that the value of $f(I)$ is incremented by one each time I assumes a value of the form $wN+1$, for $w \in Z^+$. Similarly we must determine a mapping g such that

$$K \rightarrow g(I)$$

and such that $g(I)$ assumes the successive values 1, 2,..., N, but its values wrap around each time $f(I)$ becomes $wN+1$, as it becomes evident from Figure 5.4. For the case of the loop of Figure 5.3 it can be seen that

$$J \rightarrow f(I) - \left\lceil \frac{I}{N} \right\rceil \tag{5.3}$$

$$K \rightarrow g(I) = I - N\left\lfloor \frac{I-1}{N} \right\rfloor.$$

The mappings in (5.3) satisfy the properties mentioned above. In the case of the triply nested DOALLs of Figure 5.5 the corresponding mappings are defined by,

$$J \rightarrow f(I) = \left\lceil \frac{I}{N^2} \right\rceil$$

$$K \rightarrow g(I) = \left\lceil \frac{I}{N} \right\rceil - N \left\lfloor \frac{I-1}{N^2} \right\rfloor$$

$$L \rightarrow h(I) = I - N \left\lfloor \frac{I-1}{N} \right\rfloor.$$

It is clear that the mappings f, g, and h follow a regular pattern. As it is shown below, loop coalescing can be applied to a much wider range of nested loops with unequal loop bounds. The following theorem defines the general array subscript transformation for loop coalescing. Let $L=(N_m, N_{m-1},..., N_1)$ be any m-way (non-perfectly) nested loop, and $L'=(N=N_m N_{m-1}...N_1)$ be the corresponding coalesced (single) loop. Let also $J_m, J_{m-1},..., J_1$ denote the indeces of the loops in L, and I the index of the transformed loop L'. Then we have the following.

Theorem 5.1 Any array reference of the form $A(J_m, J_{m-1},..., J_1)$ in L can be uniquely expressed by an equivalent array reference

$A(f_m(I), f_{m-1}(I),..., f_1(I))=A(I_m, I_{m-1}, \ldots, I_1)$ in L', where

$$I_k = f_k(I) = \left\lceil \frac{I}{\prod_{i=1}^{k-1} N_i} \right\rceil - N_k \left\lfloor \frac{I-1}{\prod_{i=1}^{k} N_i} \right\rfloor, \qquad (k=m,m-1,...,1) \tag{5.4}$$

or for the case of equal loop bounds,

$$I_k = \left\lceil \frac{I}{N^{k-1}} \right\rceil - N \left\lfloor \frac{I-1}{N^k} \right\rfloor, \qquad (k=m,m-1,...,1).$$

Proof: Consider an m-level nested loop L that is transformed into a single loop L' with index I,

as above. Any array reference of the form $A(J_m,..., J_i, \ldots, J_1)$ will be transformed into $A(I_m,...,I_i, \ldots, I_1)$, where I_i, $(i=m,...,1)$ are functions of I. We will derive the mapping for $J_i \rightarrow I_i$ and prove that it is given by (5.4).

A *step* is defined to be one execution of the loop-body of the innermost (1st) loop. It is clear that the 1st index I_1 is incremented by one at each step. The second index I_2 is incremented at steps of size N_1, I_3 at steps of size N_1N_2, ..., I_i is incremented at steps of size $N_1N_2 \cdots N_{i-1}$, and so on. At each moment the total number of steps (iterations) that have been completed is given by I. It is clear therefore that the expression

$$\left\lfloor \frac{I}{N_1N_2 \cdots N_{i-1}} \right\rfloor \tag{5.5}$$

is incremented by one at steps of size $N_1N_2...N_{i-1}$. However, all indeces (but the outermost) wrap around and assume repeatedly the same values for each iteration of their outermost loops. Each index assumes a maximum value which is its corresponding loop upper bound. This value is reached after N_1 steps for I_1, after N_1N_2 steps for I_2, ..., after $N_1N_2 \cdots N_i$ steps for I_i and so on. Therefore the mapping defined by (5.5) for I_i is correct as long as $I \leq N_1N_2...N_i$ but not for later steps. Thus we have to "compensate" (5.5) for the wrap around of the values of I_i. This can be done by subtracting from (5.5) the multiples of N_i at the steps at which I_i repeats its values. In other words we should subtract from (5.5) the multiples of N_i which are given by,

$$N_i \left\lfloor \frac{I-1}{N_1N_2...N_i} \right\rfloor. \tag{5.6}$$

From (5.5) and (5.6) it follows that the correct mapping for I_i is given by,

$$I_i \rightarrow \left\lfloor \frac{I}{N_1N_2...N_{i-1}} \right\rfloor - N_i \left\lfloor \frac{I-1}{N_1N_2...N_i} \right\rfloor. \quad \blacksquare$$

For the last iteration, index I_k, $(k = m, m-1,..., 1)$ should more precisely be defined by

$$I_k = \min(N_k, f_k(I)).$$

From (5.4) we also observe that for the outermost index I_m, the transformation is $\lceil I/\prod_{i=1}^{m-1} N_i \rceil$ since the second term in (5.4) is always zero.

5.1. Processor Assignment and Subscript Calculation

From a first observation, it seems that loop coalescing introduces expensive operations in the subscript expressions. Thus, one may question the practicality of such a transformation. The rather complicated subscript expressions do not pose any serious performance problem because, as it will be shown in this section, these expressions need only be evaluated once per processor, and each processor is assigned blocks of consecutive iterations. Each subscript calculation consists of two division operations, one multiplication and one subtraction.

Considering again a loop of the form $L = (N_m,..., N_1)$ all partial products $\prod_{i=1}^{j} N_i$, $(j = 1, 2,..., m)$ are obtained (and stored for later use) at no extra cost during the evaluation of $\prod_{i=1}^{m} N_i$ which involves m multiplications.

Now let us see what happens when the coalesced loop L' is scheduled on P processors. Each processor will be assigned to execute $r = \lceil N/P \rceil$ successive iterations of L'. More specifically, processor p, $(p = 1, 2,..., P)$ will execute iterations $(p-1)r + 1$ through pr of the coalesced loop.

Suppose next that an array reference of the form $A(*,...,*, f_i(I), *,..., *)$ exists in the code of L'. Then from the previous paragraphs it follows that processor p will access those elements in the i-th dimension of A that are included in

```
      DOALL 1 J=1,2
        DOALL 2 K=1,3
          DOALL 3 L=1,6

            A(J,K,L)= ....

3       ENDOALL
2     ENDOALL
1  ENDOALL
        (a)
```

becomes

```
      DOALL 1 I=1,36
```

$$A\left(\left\lceil\frac{I}{18}\right\rceil, \left\lceil\frac{I}{6}\right\rceil - 3\left\lfloor\frac{I-1}{18}\right\rfloor, I - 6\left\lfloor\frac{I-1}{6}\right\rfloor\right) = \ldots.$$

```
   1 ENDOALL
        (b)
```

becomes

```
DOALL 1 p=1,56
```

$$A\left(\left\lceil\frac{(p-1)r+1}{18}\right\rceil \ldots \left\lceil\frac{pr}{18}\right\rceil, \left\lceil\frac{(p-1)r+1}{6}\right\rceil - 3\left\lfloor\frac{(p-1)r}{18}\right\rfloor \ldots \left\lceil\frac{pr}{6}\right\rceil - 3\left\lfloor\frac{pr-1}{18}\right\rfloor, (p-1)r+1 - 6\left\lfloor\frac{(p-1)r}{6}\right\rfloor \ldots pr - 6\left\lfloor\frac{pr\ \ 1}{6}\right\rfloor\right) = \ldots.$$

1 ENDOALL
(c)

Figure 5.6. Coalescing for block scheduling.

$$A(*, \ldots, *, f_i((p-1)r+1) : f_i(pr), *, \ldots, *)$$

(where the notation $i{:}j$ denotes all increments of 1 from i to j inclusive). In general, from (5.4) it follows that the subscripts in the k-th dimension referenced by processor p are in the following interval,

$$I_k \in \left[\left\lceil\frac{(p-1)r+1}{\prod_{j=1}^{k-1} N_j}\right\rceil - N_k\left\lfloor\frac{(p-1)r}{\prod_{j=1}^{k} N_j}\right\rfloor \ldots \left\lceil\frac{pr}{\prod_{j=1}^{k-1} N_j}\right\rceil - N_k\left\lfloor\frac{pr-1}{\prod_{j=1}^{k} N_j}\right\rfloor\right].$$

In order to see in more detail how the subscript computation is performed after processors have been assigned, consider the following example. Let us suppose that we have the loop of Figure 5.6(a) that is coalesced into the single DOALL of Figure 5.6(b), which is to be executed on $P=5$ processors. In this case $N_3=2$, $N_2=3$, $N_1=6$ and therefore $r=\lceil N_3N_2N_1/P\rceil = \lceil 36/5\rceil = 8$.

Since the coalesced loop is executed on 5 processors, as far as array A is concerned, it is equivalent to the pseudo-vector loop of Figure 5.6(c). Thus, for each processor we only need to compute the value range for each subscript. Since each subscript depends only on p, all subscript ranges can be evaluated in parallel. For $p=3$ for example, the range of A that is referenced by the 3rd processor is given by,

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
A(1,1,1)	A(1,2,3)	A(1,3,5)	A(2,2,1)	A(2,3,3)
A(1,1,2)	A(1,2,4)	A(1,3,6)	A(2,2,2)	A(2,3,4)
A(1,1,3)	A(1,2,5)	A(2,1,1)	A(2,2,3)	A(2,3,5)
A(1,1,4)	A(1,2,6)	A(2,1,2)	A(2,2,4)	A(2,3,6)
A(1,1,5)	A(1,3,1)	A(2,1,3)	A(2,2,5)	
A(1,1,6)	A(1,3,2)	A(2,1,4)	A(2,2,6)	
A(1,2,1)	A(1,3,3)	A(2,1,5)	A(2,3,1)	
A(1,2,2)	A(1,3,4)	A(2,1,6)	A(2,3,2)	

Figure 5.7. Distribution of array elements (and iterations) among 5 processors.

$$A\left(\left\lceil\frac{17}{18}\right\rceil \cdots \left\lceil\frac{24}{18}\right\rceil,\ \left\lceil\frac{17}{6}\right\rceil - 3\left\lfloor\frac{16}{18}\right\rfloor \cdots \left\lceil\frac{24}{6}\right\rceil - 3\left\lfloor\frac{23}{18}\right\rfloor,\ 17-6\left\lfloor\frac{16}{6}\right\rfloor \cdots 24-6\left\lfloor\frac{23}{6}\right\rfloor\right)$$

$$\text{or} \quad A(1{:}2,\ 3{:}1,\ 5{:}6) \tag{5.7}$$

Since we know the upper bounds for each index, (5.7) uniquely determines the elements of A that will be accessed by the 3rd processor (A(1,3,5), A(1,3,6), A(2,1,1), A(2,1,2), A(2,1,3), A(2,1,4), A(2,1,5), A(2,1,6)). The detailed access pattern of the elements of A by each processor in our example is shown in Figure 5.7.

Therefore the subscript expressions that are superficially introduced by loop coalescing should not degrade performance, especially when P is small compared to the number of itera-

```
      DOALL 1 J=1,N
        DOSERIAL 2 K=1,N
          DOALL 3 L=1,N

              A(J,K,L) = ...

3       ENDOALL
2     ENDOSERIAL
1   ENDOALL
```

(a)

becomes

```
      DOSERIAL 1 K=1,N
        DOALL 2 I=1,N^2

            A(⌈I/N^2⌉,  K,  I - N⌊(I-1)/N⌋) = ...

2     ENDOALL
1   ENDOSERIAL
```

(b)

Figure 5.8. Coalescing of a hybrid loop.

tions of the coalesced loop.

Even though we have considered the most simple subscript expressions so far, it is easy to observe that loop coalescing can be applied in the same way for any polynomial subscript expression. In the following sections we generalize the transformation and show how it can be applied to hybrid and non-perfectly nested loops. In principle, loop coalescing can be used with any arbitrarily nested loop.

5.2. Hybrid Loops

Loop coalescing may be applied selectively on hybrid loops. A loop is hybrid when it contains combinations of DOALLs, DOACRs, and serial loops. An example of a hybrid loop is shown in Figure 5.8(a). In such cases loop coalescing can be applied to transform only the DOALLs of the hybrid loop. Only the subscripts of array references that correspond to the DOALLs are transformed in this case. The indeces (subscripts) of any serial or DOACR loop are left unchanged. The coalesced version of the loop in Figure 5.8(a) is shown in Figure 5.8(b).

5.3. Non-Perfectly Nested Loops, One Way Nesting

Coalescing can also be applied to non-perfectly nested loops. The subscript transformations remain the same, but care must be taken to assure correct execution of code segments that appear in different nest levels. Such code segments must be executed conditionally in the transformed loop. Let us consider for the moment only one-dimensional nesting as in the example of Figure 5.9(a), where S_1 and S_2 denote straight line code segments. Obviously if the DOALLs of the example are coalesced, segment S_1 should be executed conditionally in the transformed loop. The compiler must insert a conditional statement before the first statement of S_1. Fortunately this is an easy task for the compiler to do and the conditionals are always

```
      DOALL 1 J=1,N
          }S1
        DOALL 2 K=1,N
            }S2
2     ENDOALL
1   ENDOALL
              (a)

      becomes


      t=0
      DOALL 1 I=1,N^2
          IF (⌈I/N⌉ .NE. t) THEN
            S1
            t = ⌈I/N⌉
          ENDIF
            S2
1   ENDOALL
              (b)
```

Figure 5.9. Coalescing of a non-perfectly nested loop.

straight forward to compute.

The coalesced version of the example loop of Figure 5.9(a) is shown in Figure 5.9(b). Scalar t is a compiler generated temporary variable that is used to test the value of I and is reset each time code segment S_1 is executed. The extension to multiple nonperfectly nested loops is also straightforward.

5.4. Multiway Nested Loops

A loop is multiway nested if there are two or more loops at the same nest level. The loop in Figure 5.10(a) is a multiway (2-way) nested loop. Figure 5.10(b) shows the corresponding coalesced loop. However, extra care should be taken with multiway nested loops. As it can be

```
      DOALL 1 J=1,N
        DOALL 2 K=1,N

           A(J,K)= ....

2       ENDOALL

        DOALL 3 L=1,N

           B(J,L)= ....

3       ENDOALL
1     ENDOALL
            (a)
```

becomes

```
      DOALL 1 I=1,N^2

         A(⌈I/N⌉ ,  I - N⌊(I - 1)/N⌋) = ....
         B(⌈I/N⌉ ,  I - N⌊(I - 1)/N⌋) = ....

1     ENDOALL
            (b)
```

Figure 5.10. Coalescing of a 2-way nested loop.

observed from Figure 5.10, in this case coalescing alters the execution order of the two statements in the example. In the loop of 10(a) all elements $A(J,*)$ are computed before any element of $B(J,*)$ is computed. In the coalesced loop the order of execution changes and ordered pairs $(A(J,i), B(J,i))$ are computed for each J instead. Thus, coalescing in this case can be applied as long as the second component of the direction vector of (any) flow dependences from DOALL 2 to DOALL 3 is ">".

CHAPTER 6

OPTIMAL AND APPROXIMATION ALGORITHMS FOR HIGH LEVEL SPREADING

As mentioned in Chapter 2, there are several types of parallelism in a restructured Fortran program, but all can be characterized as vertical or horizontal. In Chapter 4 we explored loop (horizontal) parallelism, and solved the problem of compile-time processor assignment optimally. In this chapter we will concentrate on vertical parallelism and present algorithms for deterministic scheduling of vertical objects. Recall that vertical parallelism arises from the concurrent execution of lexically disjoint parts of a program.

Scheduling for vertical parallelism is also referred to as *spreading*. Depending on the granularity of the different parts of a program we have *low* and *high level spreading* for fine and coarse grain program modules respectively [Veid85]. Most instances of the spreading problem belong to the NP-Complete family of problems. In this chapter we discuss optimal solutions for some instances of high level spreading, and efficient heuristics for the intractable cases.

In Chapter 1 we discussed how Parafrase builds the data dependence graph for a given Fortran program. Recall that the data dependence graph is a directed graph with nodes representing statements of the program and arcs representing data and control dependences. The compiler can build a similar graph called the *task graph* with nodes representing higher level blocks of code such as BASs and loops, and arcs representing collections of dependences between these higher blocks. Chapter 3 discussed how basic program statements can be grouped together to form higher level blocks called *tasks*. For the purposes of this chapter we can assume that the task graph is supplied by the compiler and need not be concerned about the details of constructing such a graph.

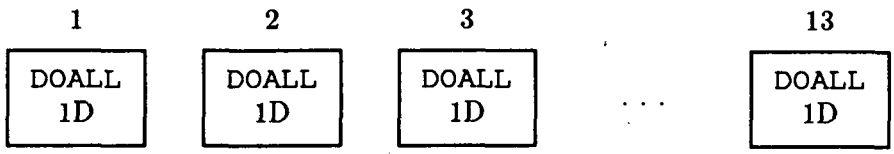


Figure 6.1. Task graph for subroutine SETDT.

Ideally, high level spreading should be applied to a set of program modules that are free of control or data interdependences. In such a case any assignment of modules to processors would be "legal" and no extra precautions need to be taken to assure correct execution. There are several such instances of independent program modules in real numerical programs. As an example Figures 6.1, 6.2, and 6.3 show the task graphs of three numerical subroutines (Denelcor benchmarks) that are commonly used in different application areas. Subroutine SETDT in Figure 6.1 consists of thirteen independent DOALL loops. The notation nD shows the dimensionality of each loop, i.e., the number of nest levels. Figure 6.2 shows a type of task graph that occurs frequently in numerical programs. The entire graph is surrounded by two serial loops. If we unroll these serial loops we get a series of uniform task graphs. Since this type of graph characterizes a large percentage of numerical subroutines we will consider them separately at the end of this chapter. Their regularity makes it easier to schedule them. Finally Figure 6.3 shows a more complex task graph (DAG) for subroutine THREEDH. Performing high level spreading for such arbitrary graphs is much more complex than for uniform graphs of the type of NOLI's. Of course we can still use simple heuristics to optimize high level spreading for random graphs locally. For example the optimal algorithm of the next section could be used to perform high level spreading for the first two levels of tasks in the graph of Figure 6.3. As we shall later see

however, optimizing schedules locally may result in non-efficient global schedules. Spreading can also be applied to lexically disjoint program modules that are "connected" with any type of dependences. For example, we can still execute concurrently the two last modules of Figure 6.3. In such a case though synchronization instructions should be used to coordinate the execution of these modules such that interdependences are satisfied in the correct order.

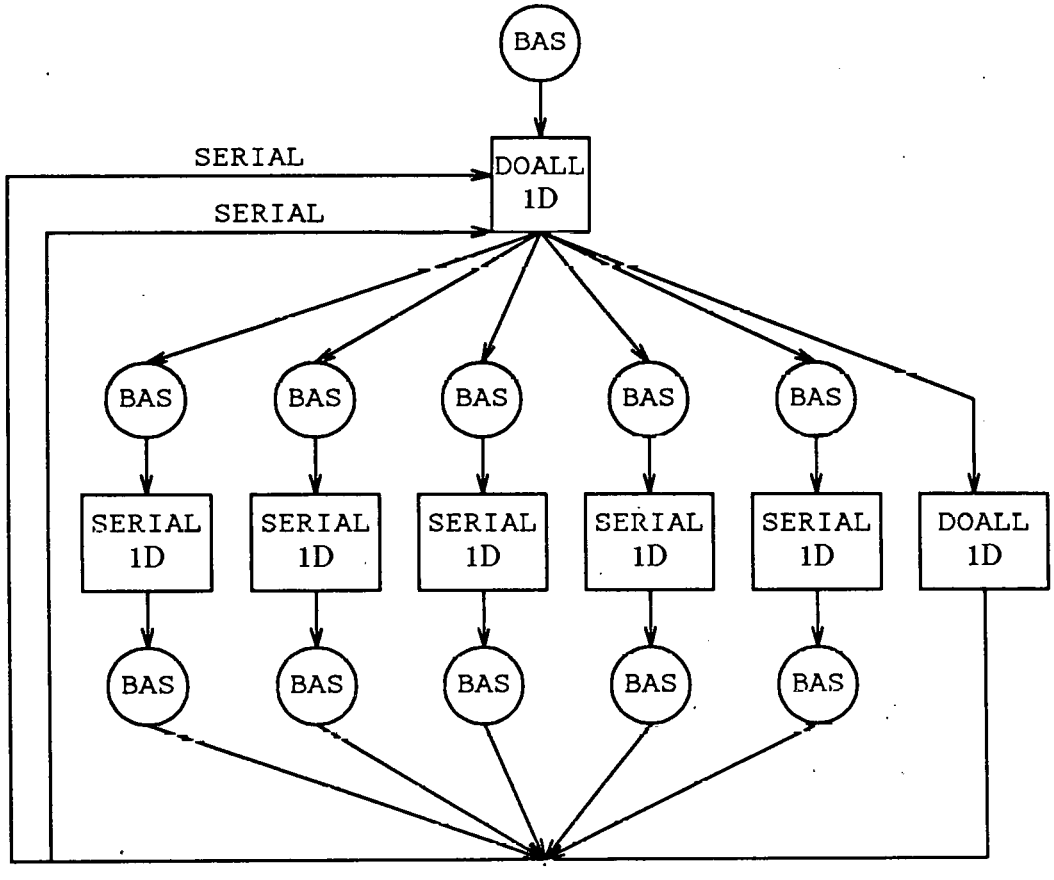


Figure 6.2. Task graph for subroutine NOLI .

In this chapter we focus on both instances of spreading, i.e., spreading of independent program modules, and spreading of program modules with interdependences. Both cases arise frequently in numerical software. First we look at the problem of spreading a set of independent tasks where the number of processors is larger than the number of tasks, each task may request any number of processors up to the maximum available, and all tasks are to be executed simultaneously. Several instances of spreading in real programs belong to this category. We solve this instance optimally in polynomial time. A fast heuristic algorithm is also presented for the case of independent tasks where the number of processors is smaller than the number of tasks and each task requests exactly one processor. This algorithm is also useful for scheduling a set of ready jobs in a multiprogramming environment where load balancing in the processors is our objective. Finally we discuss spreading of dependent tasks and scheduling of complete program task graphs. An efficient heuristic is also presented for the latter case.

6.1. Optimal Allocations for High Level Spreading

In this section we consider the instance of spreading where the number of processors is larger than the number of tasks, all tasks are to be executed concurrently, and each task may request any number of processors up to the maximum available. All tasks are independent, i.e, intertask dependences of any kind are not allowed. Let us suppose that we are given a set $S = \{M_1, M_2, \ldots, M_m\}$ of m disjoint program modules (tasks) from a given program $PROG$. Then $M_i \bigcap M_j = \varnothing$, for $i \neq j$, and $\bigcup_{i=1}^{m} M_i \subseteq PROG$. Since no dependences exist between any pair (M_i, M_j) of tasks, all elements of S may be executed simultaneously. Let us also suppose that $PROG$ is to be executed on a parallel processor system with P processors, and that each program module M_i requests $p_i \leq P$, $(i = 1, 2, \ldots, m)$ processors (the maximum it can use). In order to simplify the following discussion we always assume that $p_i \geq P$. The results that we

derive below can be trivially extended for the general case where $p_i \leq P$.

As shown below, algorithm OPTAL from Chapter 4 can be used to solve this instance of high level spreading optimally. Let T_r^i denote the parallel execution time of module M_i on r processors. We can now define the allocation function G of OPTAL for this case, and show how OPTAL computes optimal processor allocations for high level spreading. Starting from the last task M_m in S we define:

$$G_m(q) = T_q^m, \quad \text{for} \quad q = 1, 2, \ldots, P.$$

Then for $1 \leq i < m$ the allocation function (which in this case measures parallel execution time) is defined as

$$G_i(q) = \min \left\{ \max \left(T_r^i, G_{i+1}(q-r) \right) / \quad r = 1, 2, \ldots, q \right\} \tag{6.1}$$

$$\text{and} \quad (q = 1, 2, \ldots, P).$$

Following the same approach as in Chapter 4, it can be proved that $G_1(P)$ will compute the optimal allocation of P processors to the independent program modules $M_1, M_2, \ldots, M_r$. The processor allocation vector giving the optimal assignment of processors to tasks is computed similarly to Example 4.1.1 of Chapter 4. In this case however the subscripts of T^i's are used in place of the exponents of the ϵ terms in Example 4.1.1.

A point that has not been clarified yet is how we compute T_q^i for a given M_i and for different values of q. There are two alternatives for computing the parallel execution time of a task. The most precise one would be to have the compiler recompute T_q^i for each different value of q. This however may be an expensive process. A less accurate but very close (and inexpensive) approximation, would be to compute T_q^i from T_1^i, the serial execution time of M_i, as shown in (6.2).

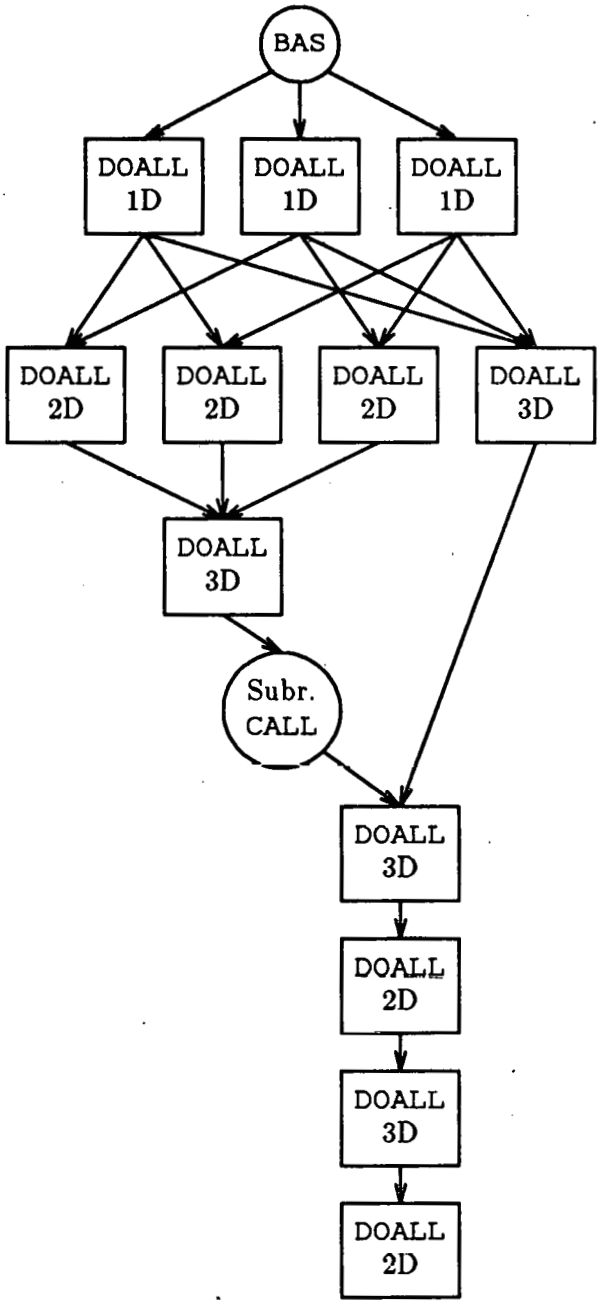


Figure 6.3. Task graph for subroutine THREEDH.

$$T_q^i = \left\lceil \frac{T_1^i}{q} \right\rceil \quad \text{for} \quad q = 1, 2, \ldots, \min(P, p_i). \tag{6.2}$$

The parallel execution time defined by (6.2) is very accurate when M_i is a loop for example, which is often the case. Each step of the algorithm involves an average of $P/2$ comparisons. There are m phases and therefore the complexity of OPTAL for high level spreading is $O(1/2mP^2)$.

When $P < m$ and $p_i = 1$, $(i = 1, 2, \ldots, m)$, i.e., when each task is allowed to execute on exactly one processor and there are more tasks than processors, the problem becomes *NP-Complete* [GaJo79]. Although this case does not occur very often during parallel processing of a single program, especially when the granularity of the tasks is fairly coarse, it arises frequently in multiprogramming environments where a set of serial jobs are to be scheduled on the processors of a system, such that processor loads are kept balanced. Since this is a special case of high level spreading at the program level we discuss it in the following section. Because of the intractability of this problem only heuristic algorithms that work in polynomial time are possible.

6.2. Scheduling Independent Serial Tasks

In this section we consider the problem of spreading a set of independent serial tasks across a number of processors in order to minimize the total execution time. Since each task may use exactly one processor, and if the number of processors is larger than the number of tasks the problem becomes trivial. We examine the case where the number of processors is smaller than the number of tasks. This problem is a classical NP-Complete problem [GaJo79] and the best known approximation algorithm so far is Multifit [CoGJ79] which uses bin-packing as its core routine. In this section we present a new heuristic algorithm for this problem that has essentially the same complexity as Multifit. Experimental results show that our algorithm outperforms

Multifit in most cases. Even though this algorithm is primarily designed for spreading a set of independent serial tasks on a parallel processor system, it can also be used for load balancing on a multiprogrammed parallel processor system where individual jobs are the unit of workload.

The *Divide-and-Fold* or D&F algorithm which is discussed below operates in two phases. Before we describe D&F in detail let us introduce the necessary definitions and nomenclature. Let us suppose that we have a set of n tasks that are ordered by execution time: $S = \{t_1 \geq t_2 \geq \cdots \geq t_n\}$. We use the execution time t_i to represent the i-th task. Our problem here is to spread the tasks in S across p processors, so that the total execution time of S is minimized. Let τ represent the execution (completion) time of S for a given distribution of tasks, i.e, the time it takes the processor with the heaviest load to process its workload. If preemption is allowed we can easily spread for the optimal τ in polynomial time. Therefore we assume that S contains nonpreemptive tasks. This is a practical restriction since, in most real cases, S consists of a set of relatively small BASs, and the overhead involved with process swapping during preemptive execution of BASs would more than eliminate the benefits of spreading.

A *list* of tasks from S is an ordered subset of tasks that are assigned to the same processor. A list of size m is represented as an m-tuple, $(t_{i+1}, t_{i+2}, \ldots, t_{i+m})$. Let

$$T = \sum_{i=1}^{n} t_i, \qquad T_{cp} = \max_{1 \leq i \leq n} \{t_i\} = t_1$$

and ω be the optimal execution time or *schedule length*. Then it is easy to prove the following lemma.

Lemma 6.1 $\tau \geq \omega \geq \max \{\lceil T/p \rceil, T_{cp}\} = LB.$

The motivation here is to generate a schedule with a length τ as close to ω as possible, by spreading the tasks of S so that the load in each processor is balanced around LB. D&F consists of two phases. During Phase I the set of tasks is partitioned into subsets of tasks and subsets are

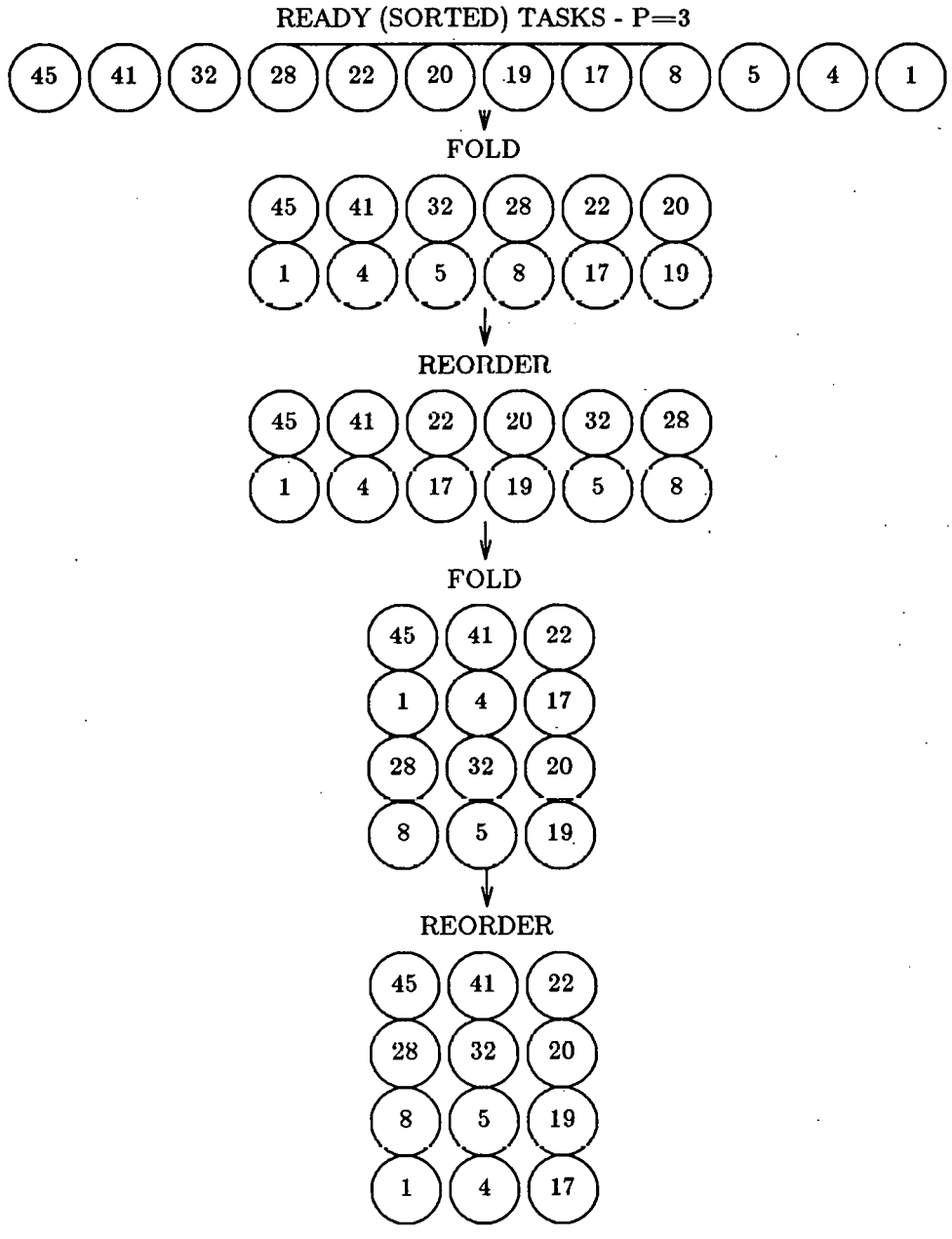


Figure 6.4: Example of the first phase of D&F.

merged together until the number of subsets is equal to p. Thus generating a set of p lists. By assigning one list to each processor we have the first "first-cut" assignment of tasks to proces-

sors. Subsequently, Phase II moves tasks between processors according to a specific procedure to obtain a finer degree of balancing. The two phases of the D&F algorithm are described below in detail.

Phase I

During this phase the set of tasks S is partitioned into $q = \lceil n/p \rceil$ subsets $S_1^1, S_2^1, \ldots, S_q^1$ with each subset but (possibly) the last containing p tasks. If q is odd, we add a "dummy" subset S_{q+1}^1 that consists of p tasks of zero execution time at the end of the list, and set $q \leftarrow q+1$. After the initial partitioning, Phase I proceeds with a series of folding steps. During the first folding we concatenate subsets S_i^1 and S_{q-i+1}^1, $(i = 1, 2, \ldots, q/2)$ such that, if $S_i^1 = \{t_1^i, t_2^i, \ldots, t_p^i\}$ and $S_{q-i+1}^1 \equiv S_j^1 = \{t_1^j, t_2^j, \ldots, t_p^j\}$, the resulting subset S_i^2 consists of p lists of size two, i.e.,

$$S_i^2 = (S_i^1, S_j^1) = \left\{(t_1^i, t_p^j), (t_2^i, t_{p-1}^j), \ldots, (t_p^i, t_1^j)\right\}$$

$$\text{for} \quad i = 1, 2, \ldots, q/2.$$

After the first folding step we have $q/2$ subsets $S_1^2, S_2^2, \ldots, S_{q/2}^2$ with each consisting of p lists of size two. Between successive folding steps we reorder the subsets by list (accumulated) execution time. At the end of each folding step an empty subset is appended if the resulting number of subsets is odd. We proceed in the same way until, after $f = \lfloor \log_2 n \rfloor$ steps, the configuration is reduced to a single subset $S_1^f \equiv S^f$ which consists of p lists. Each list contains $\lceil n/p \rceil$ tasks from the original set. Figure 6.4 illustrates the steps of Phase I for $p = 3$ and for the set of tasks shown at the top of Figure 6.4. The tasks inside each list are sorted at the end of Phase I in order of decreasing execution time. In each of the p lists we add an empty task of zero execution time needed for the tests of Phase II.

Phase II

Phase II of D&F reassigns tasks to processors selectively in order to further balance the load inside each processor, and thus reduce the overall completion time. The previous phase constructed $S^f = \{l_1, l_2, \ldots, l_p\}$ where each list l_i contains $\lceil n/p \rceil$ tasks and has been assigned to the i-th processor ($i = 1, 2, \ldots, p$). Phase II performs a single pass through the loads of the p processors considering a pair of processors $(i, p-i+1)$ at a time for ($i = 1, 2, \ldots, \lfloor p/2 \rfloor$). For each pair of processors it performs three tests and, based on the outcome of these tests, makes one reassignment. For the list l_i or processor i let T_i be the total execution time of the tasks in l_i, ($i = 1, 2, \ldots, p$). Then for each pair $(i, j=p-i+1)$ of processors perform the following: Let $\tau_1 = \max\{T_i, T_j\}$.

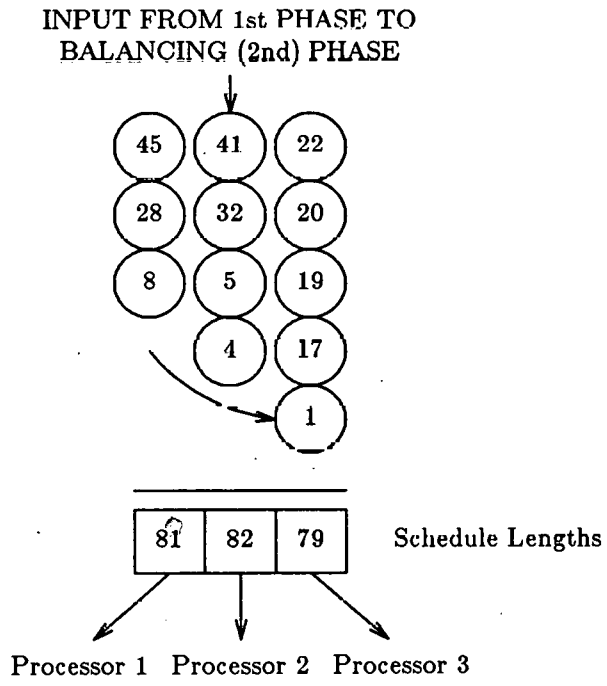


Figure 6.5: Example of the balancing (2nd) phase of **D&F**.

Test 1: Find the smallest task t_k^i of l_i for which $T_i - t_k^i \leq LB$. If such task does not exist let t_k^i be the largest (topmost) task of l_i. Let t_{k-1}^i be the next smallest task and compute

$$M1 = \max\left(T_i - t_k^i,\ T_j + t_k^i\right)$$

$$M2 = \max\left(T_i - t_{k-1}^i,\ T_j + t_{k-1}^i\right)$$

$$\tau_2 = \min(M1,\ M2).$$

Test 2: Find the smallest sum W_k^i of the first k smallest tasks in l_i for which $T_i - W_k^i \leq LB$, and let $W_{k-1}^i = W_k^i - \{t_k^i\}$. Then perform the following computations.

$$N1 = \max\left(T_i - W_k^i,\ T_j + W_k^i\right)$$

$$N2 = \max\left(T_i - W_{k-1}^i,\ T_j + W_{k-1}^i\right)$$

$$\tau_3 = \min(N1,\ N2).$$

Test 3: This test finds the optimal single exchange of tasks between the two processors. Let

$$D_i = \left\lfloor \frac{T_i - T_j}{2} \right\rfloor$$

be the difference in loads between processors i and $j = p - i + 1$. For $q = \lceil n/p \rceil$, compute

$$\tau_4 = \min\left\{ \left| D_i - (t_{k_1}^i - t_{k_2}^j) \right| \text{ for } \quad 1 \leq k_1,\ k_2 \leq q \right\}.$$

From the three tests we find the smallest value between τ_2, τ_3, τ_4 and τ_1 (that corresponds to no action) and perform the reassignment of tasks that is implied by the test for that τ. For example, if τ_2 is the minimum then if $\tau_2 = M_1$, task t_k^i is dequeued from processor i and queued in processor $j = p - i + 1$; otherwise t_{k-1}^i is reassigned to the j-th processor. If τ_3 is the minimum and $\tau_3 = N_1$, the first k smallest tasks from processor i are transferred to processor j; otherwise the first $k-1$ tasks are transferred. Finally if τ_4 is the minimum among the three, a mutual exchange of tasks between processors i and j takes place. This exchange is the one that best balances the

loads in the two processors. This rebalancing procedure is performed once for each pair of processors $(i, p-i+1)$, $(i = 1, 2,..., q)$. Figure 6.5 shows a trivial case of rebalancing for the example of Figure 6.4.

Since the tasks are ordered inside each l_i, test 3 takes an average of q comparisons for each pair of processors. The most expensive activity in Phase II is the 3rd test. Assuming that tasks are initially ordered by execution time, Phase I takes $O(log_2(n/p))$ steps to complete. Phase II of D&F is performed once but it is the bottleneck since its average complexity is $O(n^2/p^2)$. If the balancing phase is restricted to use only the first of the three tests, the complexity of Phase II is $O(p)$.

In order to test the performance of D&F against Multifit, the best known heuristic for this problem, we implemented D&F and Multifit and performed the same experiments for the two cases. The implementation of D&F has a balancing phase (II) that uses only the first of the three tests described above. Thus a full implementation of D&F as described in this chapter should perform at least as well as the current implementation. In order to compare our experiments with those for Multifit reported in [CoGJ78], the same approach was used to generate our tests. The execution times of tasks were randomly generated using normal distribution and the same size of tests as in [CoGJ78] were used.

More specifically we conducted two types of experiments. For Experiment 1 we performed 20 runs. Each run consisted of 128 tasks randomly generated with task execution times following the normal distribution with values in the range [1...100]. The 20 runs used different numbers of processors ranging from 2 to 21. Table 6.1 summarizes the results of the first experiment. $\overline{D\&F}$ and $\overline{Multifit}$ denote the average schedule lengths over all 20 runs. In the second experiment the number of processors was kept constant to p=10 and again 20 runs were performed each with a different number of tasks. The task execution times were also generated

	Wins	Losses	Ties	Optimal Schedules
D&F	19	0	1	19
Multifit	0	19	1	0

$$\frac{\overline{D\&F}}{\overline{opt.}} = 1.0005, \qquad \frac{\overline{Multifit}}{\overline{opt.}} = 1.013$$

Table 6.1. Results of 1st experiment.

	Wins	Losses	Ties	Optimal Schedules
D&F	18	1	1	14
Multifit	1	18	1	0

$$\frac{\overline{D\&F}}{\overline{opt.}} = 1.012, \qquad \frac{\overline{Multifit}}{\overline{opt.}} = 1.022$$

Table 6.2. Results of 2nd experiment.

randomly in the range [1...100]. The number of tasks in each of the twenty sets was 20, 30, 40, 50,..., 210 respectively. Table 6.2 summarizes the results of the second experiment.

So far we have discussed and presented algorithms for spreading independent serial or parallel tasks. High level spreading however can be applied to sets of tasks that exhibit inter-task dependences and thus form, in the general case, a direct graph. Such directed task graphs can be generated for a given program by the compiler. Performing spreading for a directed graph is the most difficult case of spreading, and all instances of this problem for $p > 2$ and task execution times of greater than 1 are NP-Hard [GaJo79]. In the following sections we consider high level spreading for directed task graphs and present efficient heuristics for assigning

processors to minimize execution time and maximize efficiency.

6.3. High Level Spreading for Complete Task Graphs

6.3.1. Processor Allocation for p-Wide Task Graphs

We consider here an arbitrary parallel program represented by a task graph $G \equiv G(V, E)$, where the set of nodes V represents the tasks (modules of the program) and the set of arcs E represents intertask dependences. For each such graph G we can construct its corresponding layered graph. The mechanism for deriving the layered graph of a DAG is described in Section 2.4 of Chapter 2. Since each node of the layered G may be a complex module of code it may be executable on one or more processors.

Below we present a simple linear time heuristic algorithm for allocating processors to general task graphs. We call this *Proportional Allocation* heuristic since it allocates to each node a number of processors which is proportional to the size of the node. The idea behind proportional allocation is to allocate processors to the task graph on a layer-by-layer basis, so that the load in each layer is evenly distributed across the available processors, resulting in a suboptimal execution time.

Let V_i, $(i = 1, 2, \ldots, k)$ be the layers in G and v_j, $(j = 1, 2, \ldots, n)$ the nodes of $V = \bigcup_{i=1}^{k} V_i$. Let also c_i be the cardinality (number of tasks) of layer V_i. We define the *width* of G to be the maximum number of nodes in any of its k layers. If p is the number of available processors a p-wide graph is thus a graph in which each layer contains at most p nodes. In this section we discuss high level spreading for p-wide graphs. A generalization of this algorithm that handles graphs of any width is given later in this chapter. Each node v_j of G may request $r_j \leq p$, $(j = 1, 2, \ldots, n)$ processors. For each layer V_i, $(i = 1, 2, \ldots, k)$ of G we carry out the

following steps. (The notation $x \leftarrow a$ used below indicates the assignment of an expression a to variable x.)

Step 1. Each node $v_j \in V_i$ is allocated one processor. If $|V_i| = q_i$, then the number of remain-

```
   DOALL 1 I1 = 1, 7
       } 1
1  ENDOALL

   DOALL 2 I2 = 1, 14
       } 8
2  ENDOALL

   DOALL 3 I3 = 1, 5
       } 5
3  ENDOALL

   DOALL 4 I4 = 1, 20
       } 4
4  ENDOALL

   DOALL 5 I5 = 1, 24
       } 6
5  ENDOALL
```

Number of processors allocated to each loop	
Loop Number	No. of Processors
1	1
2	9
3	3
4	7
5	12
Total	32

Figure 6.6. A simple program with DOALLs and the processor allocation profile.

ing processors is $p_R = p - q_i$. The tasks in V_i are arranged in order of decreasing size.

Step 2. The remaining $p' = p_R$ processors are allocated to the nodes of V_i with $r_j > 1$ so that each node receives a number of processors proportional to its size. For a node v_j in V_i with $r_j > 1$, the serial execution time is t_j. Let $\tau_i = \sum_{v_j \in V_i} t_j$ denote the total execution time of all nodes $v_j \in V_i$ with $r_j > 1$. Then, for all such nodes perform:

$$p_j = \left\lfloor p' * \frac{t_j}{\tau_i} \right\rfloor \tag{6.3}$$

$$p_j \leftarrow \min(r_j - 1,\ p_j) \tag{6.4}$$

$$p_R \leftarrow p_R - p_j \tag{6.5}$$

where p_j is the number of processors allocated to node v_j. Steps (6.3), (6.4), and (6.5) are repeated until all processors are allocated ($p_R = 0$), or all nodes in V_i are processed. It should be noted that if at the end $p_R > 0$, then $p_j + 1 = r_j$, $(j = 1, 2, \ldots, q_i)$. A procedural description of the proportional allocation heuristic is given in Figure 6.8. A simple example of the application of this algorithm to a single layer with DOALL loops, is shown in Figures 6.6 and 6.7. The number of processors allocated to each loop by our algorithm is shown in the table of Figure 6.6. Figure 6.7(a) shows the processor/time diagram when loops are executed one by one on an unlimited (in this case) number of processors, with a total execution time of 24 units. Figure 6.7(b) shows the processor/time diagram for the allocation performed by proportional allocation heuristic. Processors were allocated so that both horizontal and vertical parallelism are utilized; 16 units is the total execution time in this case. The total program speedup on p processors that results from the application of the above heuristic is given by Theorem 2.4 of Chapter 2.

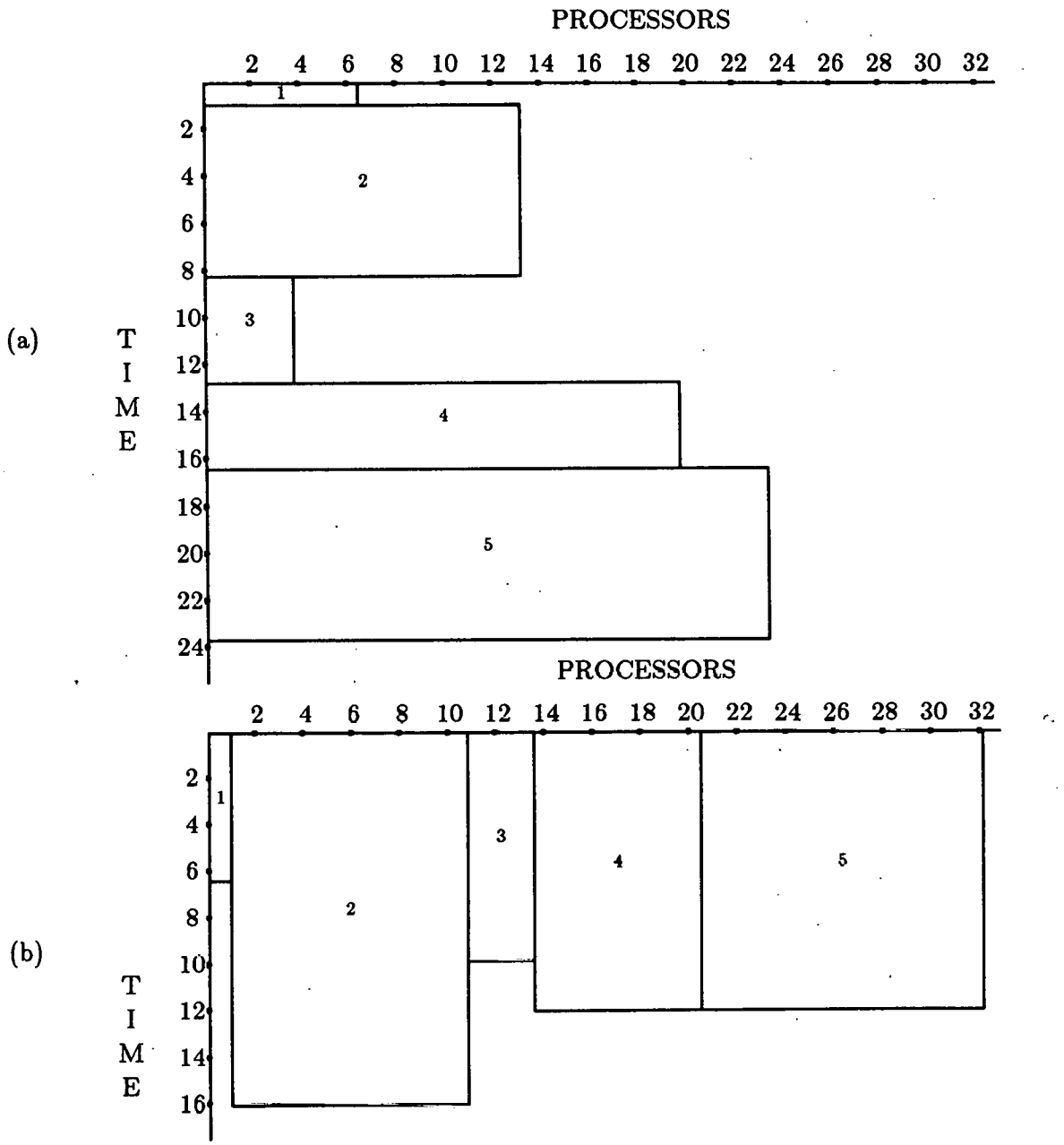


Figure 6.7. (a): The processor/time diagram for the program of Figure 2 (Case 1).
(b): The processor/time diagram after the application of the algorithm (Case 3).

P-ALLOCATION HEURISTIC

INPUT: The layered task graph $G(V, E)$ of a transformed program and the number of available processors p.

OUTPUT: A processor assignment profile for the nodes of $G(V, E)$.

FOR (All layers V_i of $G(V, E)$) DO

- Allocate one processor to each node v_j of layer V_i.
- If $q_i = |V_i|$ then set $p' = p_R \leftarrow p - q_i$, and compute $\tau_i = \sum t_j$ for all $v_j \in V_i$ with $r_j > 1$. t_j is the serial execution time of v_j and p_R the number of remaining processors.
- Sort tasks of V_i in decreasing size.

  FOR (All $v_j \in V_i$ with $r_j > 1$) AND WHILE ($p_R > 0$) DO

  - For node v_j compute:

$$p_j = \left\lceil p' * \frac{t_j}{\tau_i} \right\rceil$$

$$p_j = \min(r_j - 1,\ p_j)$$

  - Allocate p_j processors to node v_j.
  - $p_R = p_R - p_j$.

  ENDFOR

- Task $v_j \in V_i$ is allocated $p_j + 1$ processors, $(j = 1, 2, \ldots, q_i)$.

ENDFOR

Figure 6.8: The Proportional Allocation Heuristic.

6.4. List Scheduling Heuristics

The only parallel processor scheduling problem for DAGs that has been solved optimally in polynomial time is the case where $p=2$ and all tasks in the graph have unit execution time. Hardly any practical cases fall into this category. In most real cases we have $p>2$ and tasks that have varying execution times. A family of heuristic algorithms that have been developed for more general cases are the *list scheduling* algorithms [Coff76]. The most popular of them is the *critical path* heuristic. The basic idea behind list scheduling is to arrange the tasks of a given graph in a priority list and assign tasks with highest priority each time a processor becomes idle. The critical path heuristic finds the critical path of the graph, and gives priority to those tasks that compose the critical path. To find the critical path of a graph we label the nodes starting from the topmost node. Each node is labeled with the accumulated execution time of the longest path (in terms of execution time) to the first node of G.

The CP/IMS heuristic in [KaNa84] is acclaimed to be the best heuristic yet for scheduling general DAGs. This scheme is identical to the Critical Path heuristic with the following enhancement: tasks that do not belong to the critical path are given priority based also on the number of their successors i.e., the more the successor nodes of a task, the higher the priority it is assigned. As is the case with all scheduling heuristics, CP/IMS handles only graphs with serial nodes, the point being that parallel nodes can be broken down to a set of serial nodes. However this assumption is not practical. Usually program task graphs supplied by the compiler consist of a few tens of nodes. Decomposing parallel tasks even for small program graphs could create thousands of nodes that even fast heuristics could not process in a reasonable amount of time.

In the following section we discuss a scheduling heuristic that processes program graphs with parallel nodes without decomposing them. This heuristic is more general than both the critical path and the CP/IMS heuristics. Later we see how this heuristic can be coupled with

the proportional allocation heuristic to form an efficient algorithm for scheduling task graphs of any width with parallel nodes.

6.5. The Weighted Priority Heuristic Algorithm

The unique characteristic of this algorithm that distinguishes it from the CP or CP/IMS heuristics is that it covers a continuous spectrum of scheduling algorithms. In other words the *weighted priority (WP)* heuristic is a parameterized scheduling algorithm whose performance can be tuned by choosing values for a set of parameters or weights. Before we describe the WP heuristic let us see how we can construct the critical path for a task graph with parallel nodes.

Let G be a task graph with n nodes that is to be scheduled on p processors, and r_i the number of processors requested by the i-th task, (i=1, 2,..., n). An *initial* node in the graph is a task that has no predecessors and a *final* node one without successors. If a task graph has more than one initial or final node we can always change it so that it has a single initial and a single final node. This can be done by adding an empty task at the top of the graph and connecting it to all nodes without predecessors. Similarly we can add an empty final node and connect it from all nodes without successors. Starting from the initial node then we label the nodes in G visiting them in a Breadth First manner. Let t_i be the execution time of the i-th task, (i=1, 2,..., n). The initial node receives a label 0. All nodes immediately reachable from the initial node are labeled with their execution times. In general if node v_i is visited from node v_j, and x_j is the label of v_j then v_i is labeled with $x_j + \lceil t_i/r_i \rceil$. If v_i had been already labeled by an earlier visit and x_i is its old label, then the visit from v_j will relabel it with $\max(x_i, x_j + \lceil t_i/r_i \rceil)$. We continue in the same way until the final node is labeled.

To find the critical path of G we start from the bottom of the graph constructing the critical path (CP) as a set of nodes. The final node v_j is added to CP. Next we add to CP the

immediate predecessor of v_j with the largest label. We proceed in the same way until the initial node of the graph is added to the critical path. Obviously if ω_o is the optimal completion time of G on an unlimited number of processors then

$$\omega_o \geq \sum_{v_i \in CP} \left\lceil \frac{t_i}{r_i} \right\rceil .$$

The WP algorithm considers a subset of the tasks in G at a time. It computes priorities for these tasks and then allocates processors to the tasks with the highest priorities. In fact in WP "highest priority" means lowest numeric priority.

Let L_i be the set of executable tasks of G, i.e., the set of tasks that have no predecessor nodes. The algorithm schedules the tasks in L_i until all tasks of G have completed. There are k discrete steps in WP and each step processes the tasks of list L_i, $(i=1,...,k)$. The tasks of L_{i+1} cannot start executing until all tasks of L_i have completed. We want to schedule tasks so that those that constitute a bottleneck are given preference over less critical tasks. We also want to maintain a fairness criterion. In other words more processors should be allocated to tasks that are time consuming and demand many processors. The criteria are listed below and we see later how they are embedded in the WP algorithm.

Our scheme assumes nonpreemptive schedules where each task is assigned a priority, and tasks execute in ascending priority (lowest priority tasks first). The heuristic computes priorities using the following three rules of thumb:

- Give priority to tasks that belong to the critical path.
- Give priority to those tasks that have the longest execution times.
- Give priority to those tasks that have the largest number of immediate successors (i.e., break as many dependences as possible and as soon as possible).

- Give priority to those tasks that have successors with long execution times.

Below we show how to compute these three individual priorities, and from them the composite priority for each task.

At each moment during program execution, we have a set $L \subseteq V$ of runnable or executable tasks i.e., those that have no predecessors (not including the ones currently running). Suppose that the m tasks in L have execution times $t_1, t_2, \cdots, t_m$ respectively. Let a_i, $(i=1,2,...,m)$ be the number of successor tasks for each t_i, $(i=1,2,...,m)$ and $t_1^i, t_2^i, t_{a_i}^i$ be the execution times of the a_i successors of task t_i. Then we define the following:

$$A = \sum_{i=1}^{m} a_i \quad (6.6) \qquad T = \sum_{i=1}^{m} t_i \quad (6.7)$$

$$T_i = \sum_{j=1}^{a_i} t_j^i \quad (6.8) \qquad T_a = \sum_{i=1}^{m} T_i \quad (6.9)$$

and using (6.6), (6.7), (6.8), and (6.9) we define the three individual priorities in the following way (low numbers correspond to high priorities):

- Tasks with longest execution times first:

$$p_1^i = \left\lceil \frac{T}{t_i} \right\rceil \qquad (i=1,2,...,m) \qquad \text{(P1)}$$

- Tasks with largest number of successors first:

$$p_2^i = \left\lceil \frac{A}{a_i} \right\rceil \qquad (i=1,2,...,m) \qquad \text{(P2)}$$

- Tasks with largest successor-tasks first:

$$p_3^i = \left\lceil \frac{T_a}{T_i} \right\rceil \qquad (i=1,2,...,m) \qquad \text{(P3)}$$

The composite priority $P(i)$ of task t_i is then computed from (P1), (P2), and (P3) as,

$$P(i)=(\lambda_1 p_1^i + \lambda_2 p_2^i + \lambda_3 p_3^i) \qquad \text{(P4)}$$

where $0 \leq \lambda_1, \lambda_2, \lambda_3 \leq 1$ and $\lambda_1+\lambda_2+\lambda_3=1$. We call parameters λ_1, λ_2 and λ_3 the *priority weights* since they reflect the weight (significance) we give to each of these three individual priorities. Tasks on the critical path are given the highest absolute priority i.e., if L contains a task on the critical path, that task will be given priority over all other tasks in L. It is trivial to prove that at each time the set of executable tasks L contains at most one task of the critical path. This is true since, by definition, there is always a dependence between any two successive tasks (t_i, t_{i+1}) of the critical path.

After all tasks of L have been assigned a priority $P(i)$, $(i=1,2,...,m)$ for some predefined values of λ_1, λ_2 and λ_3, the processor allocation is performed as follows.

The tasks of L are ordered in increasing priority and let $L=\{t_1, t_2,..., t_m\}$ be the new order. Each t_i requests r_i processors. We choose the first k tasks from L such that

$$\sum_{i=1}^{k} r_i \leq p \leq \sum_{i=1}^{k+1} r_i.$$

Processor allocation will now be performed for the k selected tasks of L. It should be noted that if a task of the critical path belongs to L it is selected for allocation automatically no matter what its composite priority $P(i)$ is. Each of the k tasks receives one processor and the remaining $p_R=p'=p-k$ processors are allocated as shown below. Let

$$T_k = \sum_{i=1}^{k} t_i \quad \text{and} \quad p_i = \left\lfloor p' * \frac{t_i}{T_k} \right\rfloor$$

Then task t_i, $(i=1, 2,..., k)$ receives $\min(r_i - 1, p_i)$ processors and we reset $p_R \leftarrow p_R - \min(r_i - 1, p_i)$, $(i = 1, 2,..., k)$. The assignment of processors is repeated for all k tasks that were selected. If $k=m$ then we delete the nodes of L from the task graph G together with the dependences (arcs) originating from them. Those tasks that had predecessors only in L

become now the new executable tasks and are added to L. This process is repeated until all n tasks of G are assigned processors. If $k < m$ then only the first k tasks of L are deleted from G as described above. The remaining $m-k$ tasks together with any new executable tasks compose the new set L of executable tasks.

Another version of the WP heuristic is when only one priority is used to order tasks within each list as follows. Let T_1^i be the serial execution time of the i-th task in list L_j, and r_i the number of requested processors (maximum number of processors it can use). Then the priority for each task is defined as $P(i) = \lceil T_1^i / \min(r_i, p) \rceil$, where p is the number of processors in the system. Tasks are ordered inside each L_j in order of decreasing priority and processor allocation is performed in exactly the same way as above. Note that this version orders the tasks in order of decreasing minimum parallel execution time. The priorities as defined above measure the minimum possible time it takes to execute each task on p processors. This approach accomplishes two desirable goals: It gives priority to large tasks, and simultaneously, groups together tasks whose parallel execution time after proportional allocation is approximately the same (minimizing therefore idle processor time).

We can use many combinations of boundary values (0's and 1's) for the λ's to derive different heuristics. For example, if we set $\lambda_3=0$ then only task execution time and number of successors contribute to the composite priority. Another special case of this heuristic (for $\lambda_1=0$, $\lambda_2=1/2$, $\lambda_3=0$) is the CP/IMS heuristic [KaNa84] which is the best known so far. Since CP/IMS is a special case of the WP algorithm, WP performs at least as well as CP/IMS.

In the worst case where $r_i > p$, $(i=1,2,\ldots,n)$ the complexity of WP is $O(nq)$ where q is the width of the task graph. If in addition $r_i = cp$, $(i=1,2,\ldots,n)$ for some integer c, it can be shown that the WP heuristic generates the optimal processor allocation for G. In general, when the processors requested by each node are uniformly distributed the complexity of WP is $O(n)$.

6.6. Bounds for Deterministic Scheduling

In the previous sections we presented algorithms for allocating processors at compile-time to high level program modules. Only one of the algorithms is provably optimal even though the heuristics which handle the general problem generate schedules which can be very close to optimal. In this section we derive a worst case bound for any random scheduling heuristic. Coffman has shown that when dealing with serial task graphs (where nodes are serial tasks) any random heuristic can generate schedules which are at most less than twice as long as the optimal ones. In deterministic scheduling of parallel task graphs however, this worst case bound is much larger. The reason is that when we allocate several processors to several different parallel nodes, all processors should become idle before we may reassign them. (Since we process nodes in groups and in order to satisfy dependences an implicit type of barrier synchronization must be used between successive groups of nodes that are scheduled.) This may result in several idle processors in successive steps when a "purposely bad" heuristic is used. Of course intelligent heuristics should have a worst case performance which is always close to the optimal. Although the worst case bound that we prove below should not characterize any reasonable heuristic we include it for the sake of completeness.

An *atomic* operation is an indivisible operation that takes one unit of time to execute. Let *PROG* be a program that consists of n atomic operations and ω_o and ω be the optimal completion time, and the completion time of *PROG* for a specific scheduling algorithm and for p processors. Then we have the following theorem.

Theorem 6.1 $\dfrac{\omega}{\omega_o} \leq p.$

Proof Consider the extreme case where *PROG* consists of n independent atomic operations and

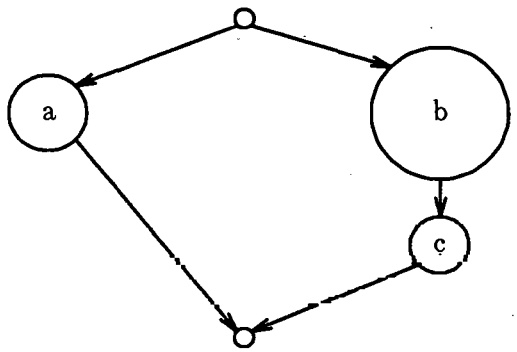


Figure 6.9. Example of a scheduling anomaly.

it is to be executed on a parallel processor system with p processors. Then it is clear that

$$\omega_o \geq \frac{n}{p} \tag{6.10}$$

Now consider how we can achieve the worst case schedule for *PROG* on p processors. Since all p processors must be utilized at least once, the worst case schedule would be the one that assigns a single atomic operation to each of the first $p-1$ processors, and the remaining of the program to the last p-th processor. The completion time of such an assignment would thus be

$$\omega \leq n - (p - 1). \tag{6.11}$$

From (6.10) and (6.11) we have

$$\frac{\omega_o}{\omega} \geq \frac{\frac{n}{p}}{n - p + 1} \quad \rightarrow \quad \frac{\omega_o}{\omega} \geq \frac{1}{p} \frac{n}{n - p + 1}$$

and for $n \rightarrow \infty$ we have

$$\frac{\omega_o}{\omega} \geq \frac{1}{p} \quad \text{or} \quad \frac{\omega}{\omega_o} \leq p. \tag{6.12}$$

Therefore the worst case schedule can be asymptotically p times longer than the optimal. ■

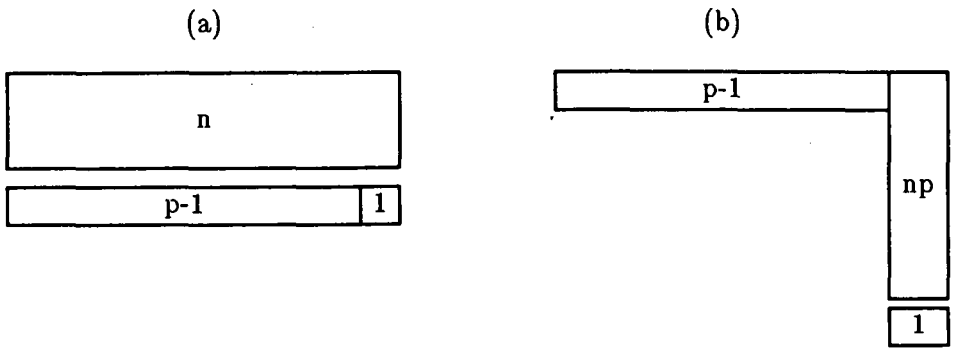


Figure 6.10. The optimal and the worst case allocation for the graph of Figure 6.10.

In fact the worst case bound of Theorem 6.1 can be asymptotically reached for non-trivial programs. Consider for example the task graph of Figure 6.9, which is surrounded by a serial loop with k iterations. Nodes a and b are parallel consisting of $p-1$ and np atomic operations respectively, and node c is a single atomic operation. Figure 6.10a shows the execution profile for the optimal allocation of G on p processors. Figure 6.10b illustrates the worst case schedule of G on p processors. From Figure 6.10 it is easy to see that $\omega_o = k(n+1)$ while $\omega = k(np+1)$. Therefore for $n \rightarrow \infty$ we have

$$\frac{\omega}{\omega_o} = \frac{np+1}{n+1} \rightarrow p.$$

Thus static scheduling heuristics that are based on local information to obtain locally optimal solutions may fail in certain cases. In the next chapter we investigate dynamic scheduling which is less sensitive in that respect.

CHAPTER 7

RUN-TIME SCHEDULING SCHEMES AND THEIR PERFORMANCE

In this chapter we concentrate on run-time environments that efficiently support scheduling and synchronization. We discuss compiler, operating system, and hardware issues. It is clear that some decisions such as resource allocation in a shared environment, memory management, transient and hard deadlock detection, should be performed dynamically at run-time (for example by the operating system or by the hardware). In this chapter we discuss run-time issues for processor allocation and synchronization that arise from different situations. We first concentrate on dynamic and self-scheduling of processors in a parallel machine during the parallel execution of a single program. Hardware and compiler support schemes are presented. The issue here as usual is performance, that is, the maximum speedup that can be achieved for a given program and for a specific parallel processor system.

7.1. Dynamic or Self-Scheduling of Processors

As discussed in the previous chapters static or compile-time scheduling is performed deterministically by the compiler at compile-time, or by the operating system at load-time before the actual execution of the program is initiated. Static scheduling is "fixed" (in terms of number of processors allocated to specific program modules) for a virtual machine with or without the same number of processors as the physical machine. The mapping of a virtual static schedule to a real schedule can be done by the operating system. Otherwise static schedules can be "fixed" for the real machine if it is known in advance that the program will execute in a dedicated environment.

In the first part of this chapter we will concentrate on non-deterministic or dynamic scheduling. A scheduling scheme is called *dynamic* when the actual processor allocation is

performed by a hardware or software emulated control unit during program execution. Therefore during dynamic scheduling decisions for allocating processors are taken "on-the-fly" for different parts of the program, as the program executes. Another form of non-deterministic scheduling scheme is *self-scheduling*. As implied by the term, there is no single control unit that makes global decisions for allocating processors, but rather the processors themselves are responsible for determining what task to execute next.

There are several factors (such as communication and task granularity) that must be taken into account during scheduling. Implementations of pure dynamic or self-scheduling could be very inefficient and involve enormous overhead. Because pure versions of these schemes will look at the instruction level for parallelism, the previous claim is substantiated. There is no method that allows us to have some knowledge about the topology of the program at run-time unless compiler support of some form is provided. Hybrid forms of dynamic or self-scheduling are possible by having the compiler "help" the control unit or the processors on making a scheduling decision. Guided self-scheduling is such a hybrid scheme that we discuss later in this chapter.

Another reason that can be used to argue against pure dynamic scheduling, is that the user may want to explicitly specify the concurrent modules of a given program and keep the task granularity within certain limits. The run-time system should support user directives as well as fully automated scheduling.

7.2. Parallel Processing with Centralized versus Distributed Control

When static (or deterministic) scheduling is used, the run-time overhead is minimal. With dynamic scheduling however, the run-time overhead becomes a critical factor and may account for a significant portion of the total execution time of a program. This is a logical consequence of dynamic or non-deterministic scheduling. While at compile-time the compiler or an intelligent

preprocessor is responsible for making the scheduling decisions, at run-time this decision must be made in a case-by-case fashion, and the time spent for this decision-making process is reflected in the program's execution time.

One way to alleviate this problem is to have the scheduler make scheduling decisions for a chunk of the program while other parts of the program are already executing on the computational processors. This however implies advanced knowledge about the structure of the program which can not be available at run-time. For example, we must know how to partition the program into a series of task sets so that during execution of a task set decisions for the scheduling of the next task set can be made "for-free". The execution time of the tasks in each task set should thus be long enough to allow the scheduler the necessary time to make the next decision. Moreover if the scheduler is the operating system the above scheduling "for free" is impossible, since one or more processors must be "wasted" to execute the operating system itself. If the scheduler is a control unit (CU), scheduling for free implies that the CU is a stand-alone unit (a superprocessor) and therefore more costly than a traditional control unit which is essentially a sequencer.

Many have argued against global control units in a parallel processor machine, stating as an argument that a single control unit will constitute a bottleneck. We do not necessarily share this view point at least from the scheduling (or concurrency control) point of view. If bottleneck is the issue rather than cost, we can argue that a sophisticated global control unit (GCU) can be designed such that it never becomes the scheduling bottleneck. The rationale behind this argument is the nature of the dynamic scheduling problem itself: No matter how many control units we have, there is always a bottleneck. Even in a fully distributed-control parallel processor system, where each processor makes its own scheduling decisions, processors must access a common pool of ready tasks. That common pool becomes the bottleneck, since it is a critical region, and

each processor has to lock and unlock semaphores to enter the critical section and grasp a task. Worse yet, there is little hope that this bottleneck can be overlapped with execution, which is possible with centralized control. One can argue that instead of a common pool of ready tasks, we can use multiple pools of ready tasks. Even in this case (unless each processor has its own pool of ready tasks) someone must make the decision on which processors access which pool at run-time. Moreover since the tasks are spawned from the same program, we should have a way of distributing them to the common pools. This argument can go on recursively but the conclusion is that due to the nature of the parallel execution of a single program, there is always some kind of bottleneck with run-time scheduling.

We will consider both powerful stand-alone GCUs and cases of distributed control. In our view the most serious drawback of a centralized control unit is its lack of fault tolerance. However there is the following soft solution to this problem as well. In any asynchronous shared memory parallel processor machine, each processor has its own control unit and can function independently of the other processors in the system. Therefore, the operating system can be designed to support hard failures of the global control unit: upon a failure of the GCU the operating system can choose one of the ordinary processors to become the new control unit. This (user transparent) transition will thus be graceful without hard failures and (perhaps) with a loss in performance. (Of course the same could happen in a fully distributed-control system, where each processor can function as a GCU.) In the case of centralized control the CU is presumably a special unit of the system with hardware modules that are not available in the computational processors. Therefore, any hard failure that can be solved by the operating system as discussed above should result in a performance degradation. This however is not a serious drawback unless the system is used for real time critical tasks where any performance degradation is untolerable; in such a case a distributed control system is necessary (unless replicated GCUs are used where

cost is not an issue compared to performance).

7.3. Design Rules for Run-Time Scheduling Schemes

It has been shown [Grah72] that in many cases of random task graphs optimal schedules can be achieved by deliberately keeping one or more processors idle in order to better utilize them at a later point. This and other scheduling "anomalies" are reported in [Grah72]. Detecting such anomalies however requires processing of the entire task graph in advance. Since this is not possible at run-time the luxury of deliberately keeping processors idle (with the *hope* that we may better utilize them later) should not be permitted. Even if we had a way of processing the task graph in its entirety at run-time, the scheduling overhead of an intelligent heuristic could be enormous in many cases.

The following guideline for any run-time scheduling scheme should always be applied: Make simple and fast scheduling decisions at run-time. This principle implicitly forbids asking (and answering) questions of the form: "How many processors should we allocate to this task?" Obviously answering such a question means, in general, that we are willing to hold up idle processors until they become as many as the number of processors requested by that task. This is exactly what we want to avoid.

Since we want to avoid deliberate idling of processors as much as possible, any run-time scheduling scheme should rather be designed to ask (and answer) questions of the following type: "How much work should we give to this processor?" In other words, when a processor becomes idle try to assign it a new task as soon as possible, making the best possible selection. As shown later this policy is guaranteed to generate a schedule length which is always by at most twice (in the worst case) as long as the optimal.

There is only one exception to the above rule. If a global control unit (GCU) is used and if the compiler is used to generate a substantial amount of scheduling information, we can decide on the number of processors for each specific task at run-time. This can be done in most cases without additional overhead assuming the GCU can make decisions about a subset of tasks while the computational processors are working on another subset. We discuss this case later in this chapter. First however we look at the more traditional dynamic scheduling where individual processors are the focus of attention instead of individual tasks.

7.4. Deciding the Minimum Unit of Allocation

Chapter 2 mentioned scheduling overhead and its impact on the granularity of parallelism that we can exploit. The scheduling overhead depends greatly on the machine characteristics (organization). So far most of the existing parallel processor systems have not addressed this issue adequately nor have they taken it into account either in the compiler or the hardware. On the CRAY X-MP for example multitasking can be applied at any level, although it has been shown that below a given degree of granularity multitasking results in a slowdown. The responsibility of multitasking a program is in addition left entirely to the user. This is also a disadvantage since the average user must know the details of the machine and the code to determine the best granularity. If the code is complex enough, e.g., containing several nested branching statements, finding the minimum size of code for multitasking would be a difficult procedure even for the most skillful programmer. In real systems where scheduling is done by the compiler or the hardware (e.g., Alliant FX/8) the critical task size is also ignored. So we may have the case where a novice programmer or the compiler schedule a single statement parallel loop on several processors with successive iterations executing on different processors. This would most likely result in a performance degradation.

Another bad, in our view, practice that has been widely adopted by users and system designers is exploiting the parallelism in DOALL loops by allocating successive iterations to different processors. Thus in a system with p processors it is common to execute a DOALL loop with $N > p$ iterations in the following way: Iteration 1 is assigned to processor 1, iteration 2 to processor 2, ..., iteration p to processor p, iteration $p + 1$ to processor 1 and so on. Therefore processor i will execute iterations i, $i + p$, $i + 2p$, ... However it is more efficient to make the assignment so that a block of successive iterations would be allocated to the same processor. For example in the above case it would be more wise to assign iterations 1, 2,..., $\lceil N/p \rceil$ to the first processor, iterations $\lceil N/p \rceil$+1, $\lceil N/p \rceil$+2,..., 2$\lceil N/p \rceil$ to the second processor and so on. Memory interleaving can not be brought up as an argument against the latter approach since memory allocation can be done to best facilitate scheduling in either case.

There are several advantages that favor the second approach of assigning iterations to processors. When iterations of a parallel loop are assigned to processors by blocks of successive iterations, each processor does not have to check the value of the loop index each time it executes an iteration. Recall that the loop index is a shared variable and each processor must lock and unlock a semaphore in order to be granted access to it and get the next iteration. In case all processors finish simultaneously they will all access the loop index serially going through a time consuming process. In the worst case N accesses to a shared variable will take place. If the assignment of blocks of iterations is performed instead, only p accesses to the shared loop index will be done in the worst case. For a large N and a small p this will result in a substantial savings, considering the fact that each access to the loop index will have to go through the processor-to-memory network. Note that the number of accesses to the loop index is independent of N in our case. Another advantage of this scheme is that when we execute FORALL loops in parallel, the block assignment can be done so that the cross-iteration dependences are con-

tained within one block and the dependences are therefore satisfied by virtue of the assignment. In what follows the second method is used, that is, whenever a parallel loop (excluding DOACRs) is executed on several processors, the allocation will be done so that each processor is assigned a block of successive iterations.

Another point of interest here is the difference in scheduling flexibility between DOALL and DOACR loops. By definition, the iterations of a DOALL loop can be scheduled and executed in any order. For example a DOALL may be scheduled vertically (in which case blocks of consecutive iterations are assigned to the same processor), or horizontally (where consecutive iterations are assigned to different processors). Any permutation of the index space is legal in the case of DOALLs. By contrast, a DOACR loop can be scheduled only horizontally. In a DOACR loop there are cross-iteration dependences between any pair of consecutive iterations. Thus vertical scheduling of a DOACR amounts to essentially executing that loop serially. This fundamental difference between DOALLs and DOACRs should be taken carefully into consideration during implementation of scheduling on parallel machines that support both types of loops.

Let us return to the main subject of this section, namely critical task size. As mentioned earlier in this chapter our philosophy about run-time scheduling is to put the emphasis on the amount of work that we assign to an idle processor, rather than on the number of processors assigned to a particular loop. It should be clear by now that the former approach is more appropriate and practical for run-time scheduling, while the latter is more appropriate for compile-time scheduling. The obvious problem however is that at run-time we can not afford calculations to estimate the projected execution time of a task, and then decide how it should be partitioned so that the basic partitions are "large enough". The obvious answer to this problem is the compiler. The compiler can accomplish the same with no less accuracy. Estimating the projected execution time of a piece of code (on a single processor) can be done by the compiler or

```
1:  B1
    if C1 then B2
    else B3
    B2
    if C2 then goto 1
    else if C3 then B4
         else B5
    exit
    B3
    if C4 then B6
    else B7
```

Figure 7.1. An example of conditional code.

the run-time system with the same precision.

Let us take for example the case of a DOALL loop without conditional statements. All that needs to be done is estimate the execution time of the loop body, and let it be B. The exact number of loop iterations need not be known at compile-time. Usually the scheduling overhead is constant for a particular machine and it may only depend on the code characteristics. For example, if instruction fetching is considered to be part of the overhead, loops would have different overhead than high level spreading. With loops each processor receives the same set of instructions which is broadcast to all processors; with high level spreading each processor will receive a different set of instructions and the network traffic would thus be higher. In any case, the estimate for the execution time of a piece of straight line code can be done as precisely by the compiler as by the run-time system (operating system of hardware). Since we know the overhead for the particular machine and the particular case we can find the critical block size for that DOALL that is, the minimum number X of iterations for which $T_p < T_1$. We see how X can be determined in the next section. This number X can be "attached" to that DOALL loop as an attri-

bute at compile-time. All the run-time system must do during execution, is to assign to an idle processor X or more iterations of that particular loop (but no less). In case where $X \leq N$ the loop is treated as serial.

The same technique can be used with high level spreading in the absence of conditional statements. When high level spreading involves only basic blocks (Chapter 1) and no loops, any conditional statement can be executed in the GCU, thus following the correct execution path without significant overhead. A more difficult problem is determining the critical task size of parallel loops with conditional statements at run-time. Again we have the same limited choices (in the compiler or in the run-time system) to determine accurately the critical task size. Again the compiler can solve the problem by applying a more conservative approach.

Let us consider the code inside a DOALL loop excluding the loop statements. The control-flow graph of a code module with conditional statements can be uniquely represented by a pseudo-tree. Consider for example the code module of Figure 7.1 which constitutes the loop body of some DOALL. The corresponding control-flow graph is shown in Figure 7.2. Since there is no hope of accurately estimating the execution time either in the compiler or at run-time in this case, we choose to follow a conservative path. The execution time of each basic block $B_1, \ldots, B_7$ can be estimated precisely. We take the execution time of the loop body to be equal to the execution time of the shortest path in the tree.

The shortest path can be found by starting from the root of the tree and proceeding downwards labeling the nodes with the following procedure. Let t_i be the execution time of node (basic block) v_i, and l_i be its label. The root v_1 is labeled t_1. Then a node v_i with parent node v_j is labeled with $l_i = l_j + t_i$. As we proceed we mark the node with the minimum (so far) label. In case we reach a node that has already been labeled (cycle) we ignore it. Otherwise we

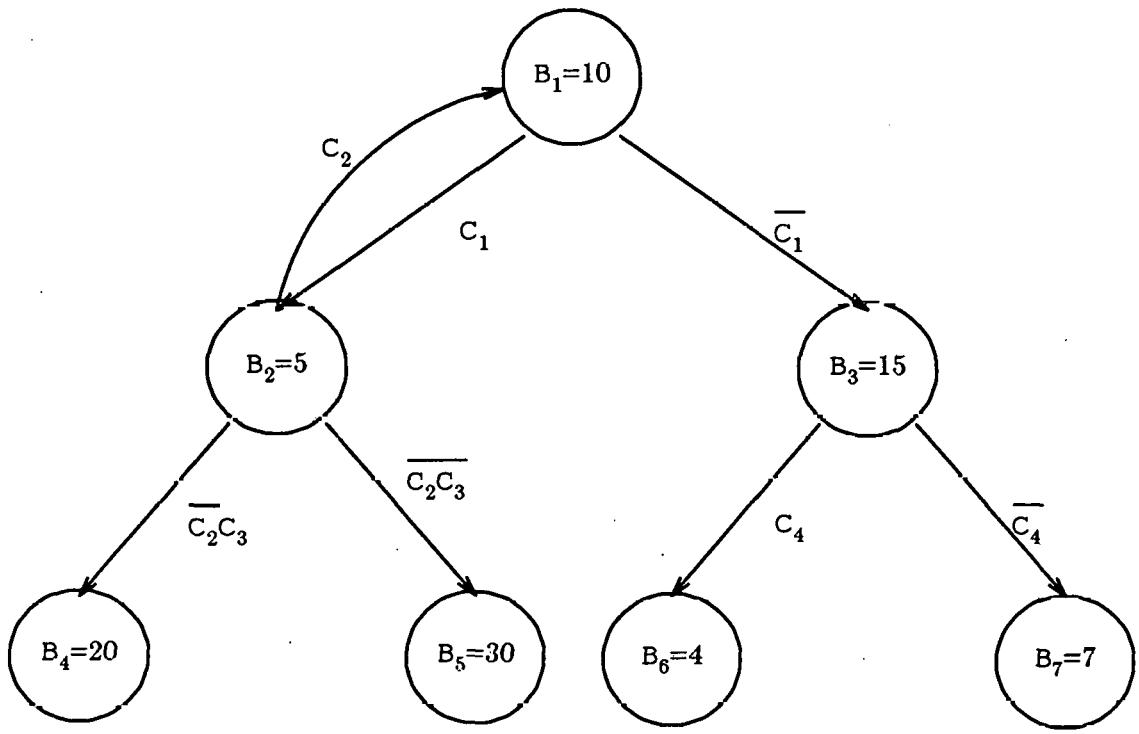


Figure 7.2. The control flow tree for the example of Figure 7.1.

proceed until we reach the leaves of the tree. Note that the labeling process does not have to be completed: If at some stage of the algorithm the node that has the minimum label happens to be a leaf, or it leads to an already labeled node (cycle) the labeling process terminates. The path π that consists of the marked nodes is the shortest execution path in that code. The number of iterations required (conservatively) to form the "critical mass" is a function of the number of processors as shown in the following section. B, the execution time of π, is given by the label of the last node of path π. A less conservative approach would be to take the sum of the execution times of all paths in the tree and divide it by the number of leaves and cycles. In the example of Figure 7.2 the above procedure gives us $B = 15$ and $B = 33.33$ respectively. All the above

can be easily implemented in the compiler.

To summarize the above we assume that the critical task size is always supplied by the compiler and is observed (from below) by the run-time system during execution. For the case of a loop which is the most important case, the critical task size is specified to the run-time system as a loop attribute which is constant and specifies the minimum number of iterations that can be allocated each time. In the case of high level spreading the compiler can easily group basic blocks together to form tasks that always meet or exceed the critical task size.

7.5. Run-Time Scheduling Overhead and Its Impact on Parallelism

During the last few years the field of parallel processing has undergone an enormous growth. Parallel processor machines with thousands and millions of processors, have often been the subject of research and development projects in industrial and academic laboratories. Looking at what has happened and what is planned for the next few years we see that general purpose parallel processor systems have been restricted to a few processors only. Cost, or the absence of applications that can use thousands of processors is clearly not the reason. The overhead involved with the simultaneous application of many processors to the same task can be enormous. Moreover we have not yet developed algorithms and methods to coordinate efficiently many processors that work on the same problem. The only exception is special purpose machines that have been built for specific applications, and can utilize large numbers of special purpose processors by using a fixed task assignment policy.

In order to make it feasible to build very large-scale parallel processor systems we must solve first the scheduling and the run-time overhead problems. In this section we study analytically two widely accepted models of overhead and their impact on the degree of parallelism that we can exploit. When a parallel task is distributed to several processors at run-time, it incurs a

penalty or overhead that limits the degree of task granularity. Consider for example the parallel execution of a DOALL loop whose iterations are spread across processors at run-time. Self-scheduling for example or any other dynamic scheduling scheme falls into this category. Run-time overhead may include several activities that do not occur during the serial execution of a loop. All processors involved for example will have to access the ready task queue in a serial mode since it is a critical section. Different processors will get different iterations of the same loop. At the end of the loop all processors involved must "pass through a barrier" serially to determine that the loop has been executed and that they are allowed to proceed with the next task. Even during execution the processors may have to access several shared variables as is the case for example, with DOACR loops. The fetching of instructions at run-time can also be considered part of the overhead. Especially with self-scheduling, instructions fetches cannot be overlapped with execution since by definition, it is impossible to predict what part of the program or which iteration of a loop a given processor will execute next. All these activities prolong the parallel execution time of a program. None of the above occurs during serial execution. This overhead, as would be expected, makes it inefficient to execute in parallel small tasks or to use a very large number of processors on even large parallel tasks. If the task is not large enough to amortize the overhead, we may end up with a parallel execution time which is larger than the serial execution time.

In this section we study the overhead involved with parallel tasks and its impact on the maximum degree of "usable" parallelism. In our case a parallel task is a parallel loop or a set of independent serial modules of a program. We analyze the case of parallel loops and the same is applicable to high level spreading. The tasks involved in an instance of high level spreading can be thought of as iterations of a DOALL loop whose loop-body contains conditional statements, and therefore different iterations have different execution times. Therefore high level

spreading can be reduced to the parallel loop case where the number of iterations equals the number of independent tasks in that set. Since it is impossible to precisely estimate the execution time of a loop body with conditional statements, either at compile-time or at run-time, we assume an average or a worst case value as mentioned in the previous section. For the moment let us assume that the loop-body for a given parallel loop has a constant execution time.

To analyze the run-time overhead we use two different widely accepted conjectures that have been backed by empirical results. The first conjecture states that during the parallel execution of a parallel task the run-time overhead is linearly proportional to the number of processors involved. The second conjecture states that advanced techniques and special hardware modules can be used to make the run-time overhead logarithmically proportional to the number of processors. We therefore have $O(p)$ and $O(log_e p)$ overhead respectively. In what follows we analyze each position and derive practical results for each case. We develop the two models and derive the optimal number of processors that can be used for a given parallel task. The two models were implemented and quantitative results are presented in section 7.5.3 for a few randomly selected parallel loops.

The results presented in this section are practical since they can be used by the compiler to draw exact or approximate conclusions for each task in a program, and used at run-time to avoid inefficient allocations. The analytical results are also applicable to other cases, for instance to study the overhead involved when several processors access a shared variable, or study the effect of combining memory requests in network switches.

7.5.1. Run-time Overhead is $O(p)$

As mentioned above we can always identify a parallel task with a DOALL without loss of generality. Let as usual T_1 and T_p denote the serial and parallel execution time of a given task.

Let N be the number of iterations of a loop and B the execution time of the loop-body. If the loop-body has a varying execution time the procedure of section 7.4 can be used to derive a worst case or average value for B.

In this section we consider the case where the run-time overhead is linearly proportional to the number of processors assigned to that loop. Let σ_o be the run-time overhead constant which in general depends on the characteristics of the code. The compiler can supply the value of σ_o for each loop (parallel task) in the program. Obviously the serial execution time of a loop with N iterations and a loop-body execution time of B would be $T_1 = NB$. The parallel execution time then on p-processors would be

$$T_p = \left\lceil \frac{N}{p} \right\rceil B + \sigma_o p. \tag{7.1}$$

Consider (7.1) as a function of p. If overhead was zero, (7.1) would be an integer valued monotonically decreasing function. Since (7.1) is not a continuous function it is not amenable to analytical study. We can easily approximate the function in (7.1) by a continuous function, by eliminating the ceiling. We thus get

$$T(p) = NB/p + \sigma_o p. \tag{7.2}$$

(7.2) is a continuous real function in the interval $(0, +\infty)$, with continuous first and second derivatives. We can thus study its shape and determine the point where overhead becomes minimal. In other words we want to find the value of p for which (7.1) becomes minimum and therefore the speedup of that task is maximized. The result is given by the following theorem.

Theorem 7.1 The parallel execution time of a task that consists of N serial processes, each taking B units of time to execute, is minimized when the task is executed on a number of processors given by

$$p_o = \sqrt{NB / \sigma_o} \tag{7.3}$$

Proof First we show how (7.3) is derived and then prove that it is indeed the optimal value for that task (loop). Consider (7.2) which is an approximation to the parallel execution time defined by (7.1). $T(p)$ is continuous in the interval $(0, +\infty)$ and has a first derivative

$$\frac{dT(p)}{dp} = T'(p) = -\frac{NB}{p^2} + \sigma_o. \tag{7.4}$$

The local extreme points of (7.2) are at the roots of its first derivative.

$$p_{o,1} = \pm\sqrt{NB / \sigma_o} \tag{7.5}$$

and since we are only interested for values in the interval $(0, +\infty)$, we discard the negative root p_1. The second derivative of $T(p)$ is

$$\frac{d^2T(p)}{dp^2} = T''(p) = \frac{2NB}{p^3} > 0 \tag{7.6}$$

(7.6) is always greater than zero and therefore the extreme at $(p_o, T(p_o))$ is a minimum, where p_o is given by (7.5). If p_o is an integer, then the parallel execution time T_p is also minimized as we prove later, and it is

$$T_{p_o} = \frac{NB}{\sqrt{NB/\sigma_o}} + \sigma_o\sqrt{NB / \sigma_o} = \sqrt{NB\sigma_o} + \sqrt{NB\sigma_o} = 2\sqrt{NB\sigma_o}. \tag{7.7}$$

Indeed if T_p is the parallel execution time for any other $p = c\sqrt{NB/\sigma_o}$ where c can be of the type m, or $1/m$ where m is a positive integer, then $T_{p_o} < T_p$, or equivalently,

$$2\sqrt{NB\sigma_o} < \sqrt{(NB)^2\sigma_o / c^2(NB)} + \sqrt{c^2\sigma_o^2(NB) / \sigma_o} \tag{7.8}$$

and if we substitute $x = NB\sigma_o$ in (7.8) we have

$$2\sqrt{x} < \sqrt{x / c^2} + \sqrt{c^2 x} \rightarrow 0 < x(1 + c^4 - 2c^2)$$

and since $x > 0$, we finally get $(1 - c^2)^2 > 0$ which is always true. ■

Corollary 7.1 The parallel execution time T_p is also minimized when the number of processors is $p_o = \sqrt{NB/\sigma_o}$.

Proof Let us suppose that there is another $p = p_o + k$, where k is a positive or negative integer, for which $T_p < T_{p_o}$. Then from (7.1) we have,

$$\left\lceil \frac{N}{p_o + k} \right\rceil B + \sigma_o(p_o + k) < \left\lceil \frac{N}{p_o} \right\rceil B + \sigma_o p_o$$

$$\text{or} \quad B\left(\left\lceil \frac{N}{p_o} \right\rceil - \left\lceil \frac{N}{p_o + k} \right\rceil \right) > \sigma_o k. \tag{7.9}$$

But from Theorem 7.1 we know that

$$\frac{NB}{p_o + k} + \sigma_o(p_o + k) > \frac{NB}{p_o} + \sigma_o p_o$$

$$\text{or} \quad B\left(\frac{N}{p_o} - \frac{N}{p_o + k} \right) < \sigma_o k. \tag{7.10}$$

By dividing the two sides of (7.9) and (7.10) we get

$$\frac{B\left(\left\lceil \frac{N}{p_o} \right\rceil - \left\lceil \frac{N}{p_o + k} \right\rceil \right)}{B\left(\frac{NB}{p_o} - \frac{NB}{p_o + k} \right)} > \frac{\sigma_o k}{\sigma_o k}$$

and after the simplifications we finally have

$$\left\lceil \frac{N}{p_o} \right\rceil - \frac{N}{p_o} > \left\lceil \frac{N}{p_o + k} \right\rceil - \frac{N}{p_o + k} \tag{7.11}$$

and since, by definition, $\lceil x \rceil - x \leq 1$ for any real x, (7.11) gives us $1 > 1$. Therefore the initial hypothesis can never be true for any integer k and thus p_o is optimal for (7.1) also. ■

Corollary 7.2 For $\sigma_o \geq \frac{NB}{4}$ the approximation function $T(*)$ defined in (7.2) satisfies

$$T(p) \geq NB \tag{7.12}$$

for any integer $p \neq 0$.

Proof By substituting $T(p)$ from (7.2) in (7.12) we have

$$\frac{NB}{p} + \sigma_o p > NB \quad \text{or} \quad \sigma_o p^2 + p(NB) + NB > 0. \tag{7.13}$$

(7.13) is a quadratic equation of p and since $\sigma_o > 0$, the inequality in (7.13) is always true if the determinant D of the equation in (7.13) is negative, i.e.,

$$D = (NB)^2 - 4\sigma_o(NB) < 0 \quad \text{which gives us} \quad \sigma_o > \frac{NB}{4}. \blacksquare$$

Corollary 7.3 If $\sigma_o \geq \frac{NB}{k}$ then the parallel execution time for $p \geq k$ is greater than the serial execution time, i.e., $T_p > T_1$.

Proof The proof here is trivial. Suppose that for some positive $p > k$ we have $T_p \leq T_1$. Then $\lceil N/p \rceil B + \sigma_o p \leq NB$, and from the statement of the corollary we have

$$\left\lceil \frac{N}{p} \right\rceil B + p \frac{NB}{k} < NB \quad \text{or} \quad 0 < \left\lceil \frac{N}{p} \right\rceil < N(1 - \frac{p}{k})$$

and since N is always positive it should be $p < k$ which contradicts the statement. ■

In the next section we analyze the case of logarithmic overhead in a similar way.

7.5.2. Run-Time Overhead is $O(log_e p)$

Let us assume that the run-time overhead is logarithmically proportional to the number of processors assigned to a parallel task. Again the task consists of N independent serial processes where each process takes B units of time to execute. In this case therefore, the parallel execution time is given by

$$T_p = \left\lceil \frac{N}{p} \right\rceil B + \sigma_o \log p. \tag{7.14}$$

To determine the optimal number of processors that can be assigned to a given parallel task in

this case, we follow the same approach as in the previous section. Again since (7.14) is not a continuous function we approximate it with

$$T(p) = \frac{NB}{p} + \sigma_o \log p \tag{7.15}$$

where $T(p)$ is now a continuous function in the interval $(0, +\infty)$, with continuous first and second derivatives. The corresponding theorem follows.

Theorem 7.2. The approximate parallel execution time defined by (7.15) is minimized when

$$p_o = \frac{NB}{\sigma_o}.$$

Proof The first derivative of (7.15) is given by

$$\frac{dT(p)}{dp} = T'(p) = -\frac{NB}{p^2} + \frac{\sigma_o}{p}. \tag{7.16}$$

$T'(p)$ has roots $p = 0$, which is discarded, and

$$p_o = \frac{NB}{\sigma_o}. \tag{7.17}$$

The second derivative of (7.15) at p_o is

$$T''(p) = \frac{2NB - \sigma_o p}{p^3} \quad \text{and} \quad T''(NB/\sigma_o) = \frac{\sigma_o^3}{(NB)^2} > 0.$$

Therefore $T(p)$ has a minimum at $p = p_o$. ■

If p_o is an integer, then we have the following corollary.

Corollary 7.4. For any $p = p_o + k$, where k is any integer, we have $T_{p_o} < T_p$.

Proof The proof is similar to the proof of Corollary 7.2. Again we assume that there is a k for which $p = p_o + k$ and $T_{p_o} > T_p$, i.e.,

$$B\left(\left\lceil\frac{N}{p_o}\right\rceil - \left\lceil\frac{N}{p_o + k}\right\rceil\right) > \sigma_o \log\left(\frac{p_o + k}{p_o}\right). \tag{7.18}$$

But from Theorem 7.2 we have that

$$B\left(\frac{N}{p_o} - \frac{N}{p_o + k}\right) < \sigma_o \log\left(\frac{p_o + k}{p_o}\right). \tag{7.19}$$

Dividing (7.18) and (7.19) and using again the fact that $\lceil x \rceil - x \leq 1$, we reach a contradiction to the initial hypothesis. Therefore the statement of the corollary is true. ■

If p_o is not an integer, then we have the following theorem.

Theorem 7.3. Let $\epsilon = \lceil p_o \rceil - p_o$ where $0 < \epsilon < 1$. Then the number of processors p_o' that minimizes the parallel execution time T_p in (7.14) is given by

$$p_o' = \begin{cases} \lfloor p_o \rfloor & \text{if } \epsilon \leq 0.5 \\ \lceil p_o \rceil & \text{if } \epsilon > 0.5 \end{cases} \tag{7.20}$$

where $p_o = NB / \sigma_o$.

Proof Let us suppose that the optimal execution time T_p is achieved for $p = p_o' + k$, where k is some nonzero integer and $|k| \neq 1$. Then

$$T_p < T_{p_o'}. \tag{7.21}$$

But in Theorem 7.2, we proved that $T(p)$ is a parabola and reaches its minimum value at $p_o = NB / \sigma_o$. From calculus we know then that for $|\xi_1| < |\xi_2|$ we have

$$T(p_o - \xi_1) < T(p_o - \xi_2)$$

and since $\epsilon < k$ and $1 - \epsilon < k$ we have

$$T(p_o \pm \epsilon) < T(p_o \pm k) \quad \text{and} \quad T(p_o \pm (1 - \epsilon)) < T(p_o \pm k)$$

or equivalently,

$$T(p) > T(p_o'). \tag{7.22}$$

Using the same approach as in the previous corollary and substituting the T_p's and $T(p)$'s in (7.21) and (7.22) we contradict the initial hypothesis. Therefore the statement of the theorem is true. ■

Corollary 7.5. If $\sigma_o \geq \frac{NB(p-1)}{p}$ then $T_p > T_1$.

Proof Suppose that $T_p < T_1$. Then

$$\frac{N}{p}B + \sigma_o \log p \leq \left\lceil \frac{N}{p} \right\rceil B + \sigma_o \log p \leq NB. \tag{7.23}$$

By taking the first and last terms in (7.23) we have

$$NBp - NB - \sigma_o \log p > 0. \tag{7.24}$$

Let $NB = \log y$. Then $NBp = p \log y = \log y^p$ and (7.24) becomes

$$\log y^p - \log y - \log p^{\sigma_o p} > 0, \quad \text{or}$$

$$\log \left(\frac{y^p}{y p^{\sigma_o p}} \right) > 0 \quad \text{or} \quad \log \left(\frac{y^{p-1}}{p^{\sigma_o p}} \right) > 0$$

and finally,

$$y^{p-1} > p^{p\sigma_o} \quad \text{or} \quad e^{NB(p-1)} > p^{p\sigma_o}$$

and since $p > e$ we finally have that

$$NB(p-1) > p\sigma_o \quad \text{or} \quad \sigma_o > \frac{NB(p-1)}{p}$$

which contradicts the basis of the corollary and thus (7.23) is not true. ■

7.5.3. Measurements

We used the above models to study the effect of run-time overhead on the degree of usable parallelism and thus on execution time. We used (7.1) and (7.2) to compute the actual execution time of a program, and (7.2) to compute its approximation function for the linear overhead case.

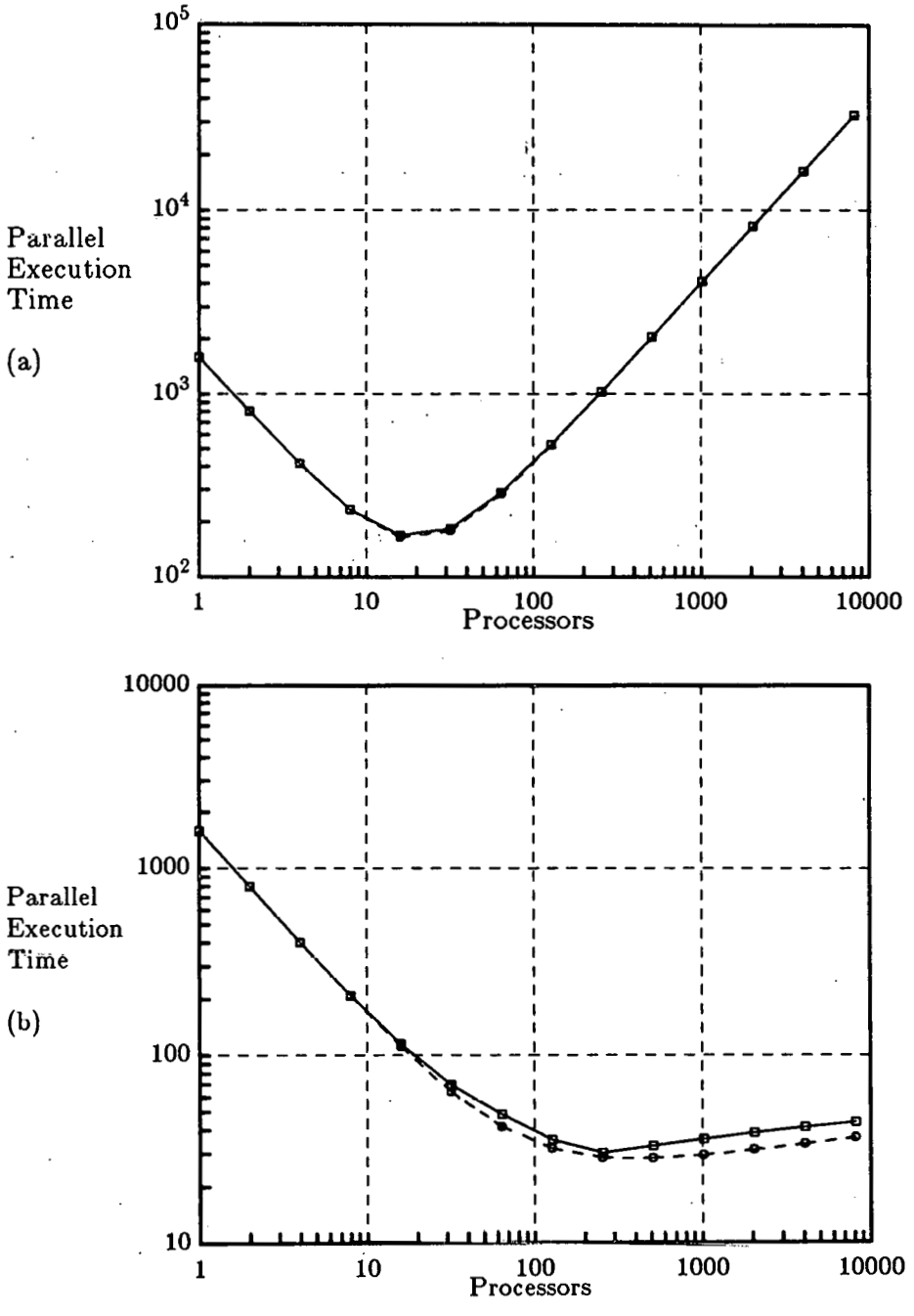


Figure 7.3. Execution times with (a) linear and (b) logarithmic overheads for N=200 and B=8.

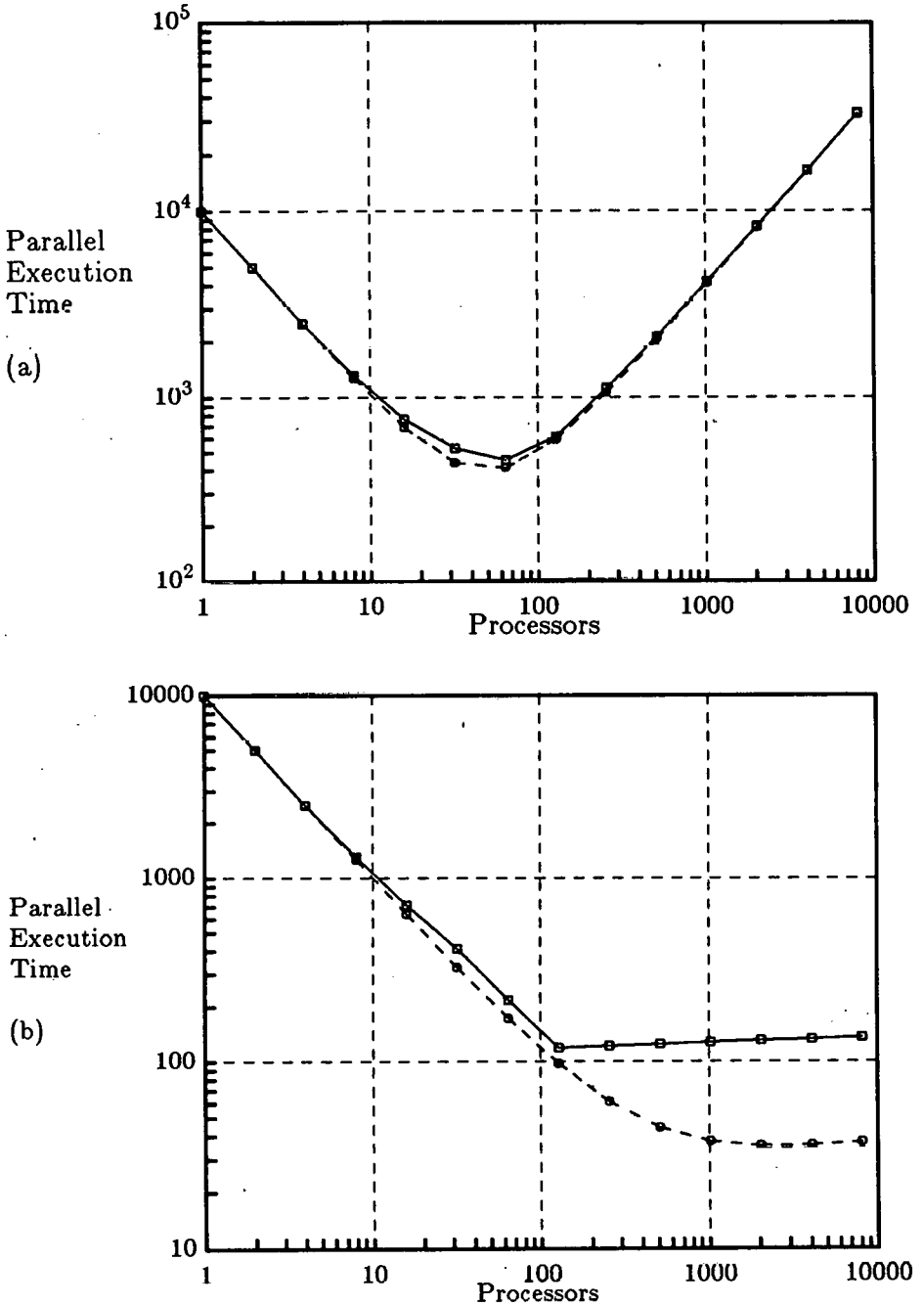


Figure 7.4. Execution times with (a) linear and (b) logarithmic overheads for N=100 and B=100.

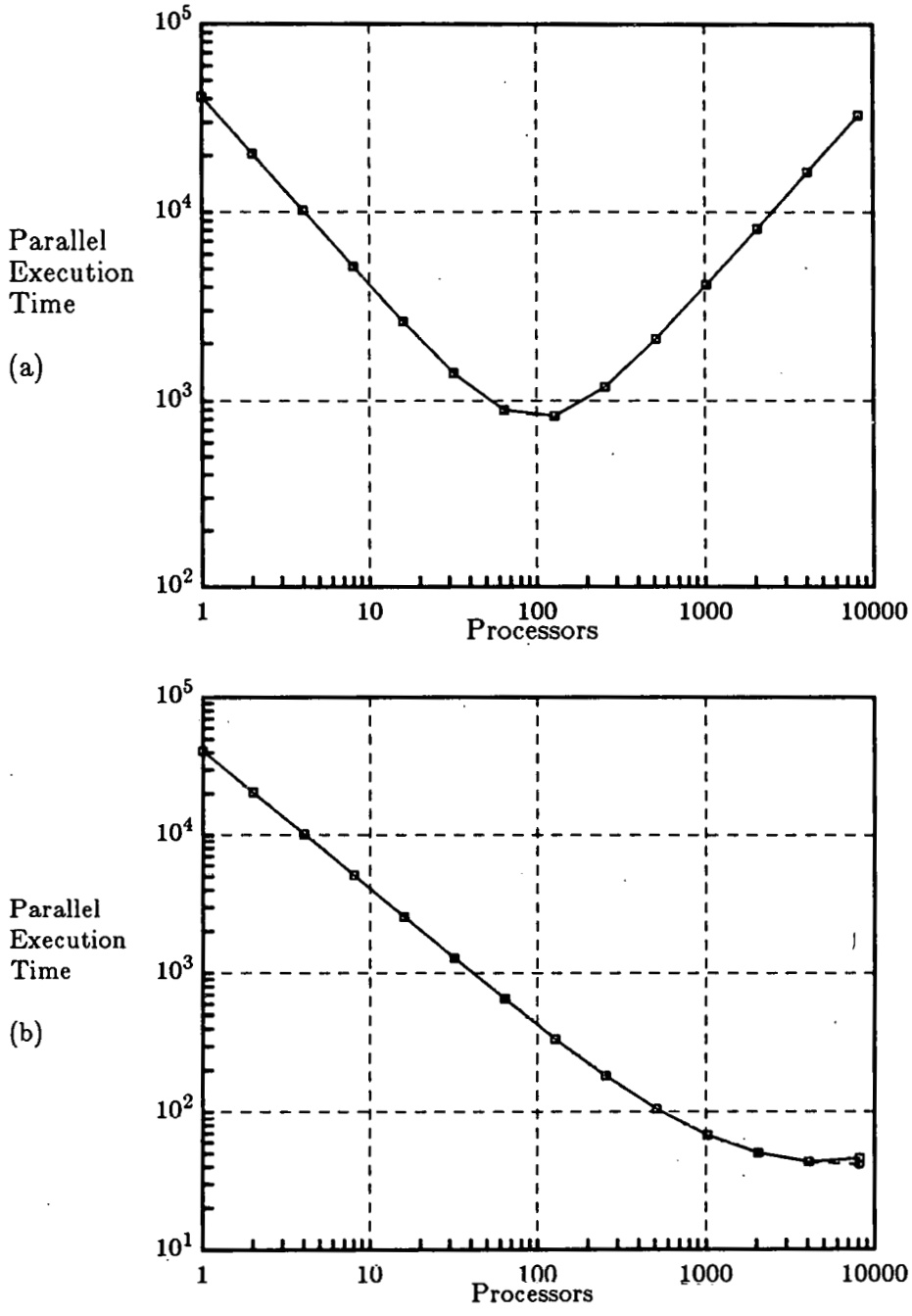


Figure 7.5. Execution times with (a) linear and (b) logarithmic overheads for N=4096 and B=10.

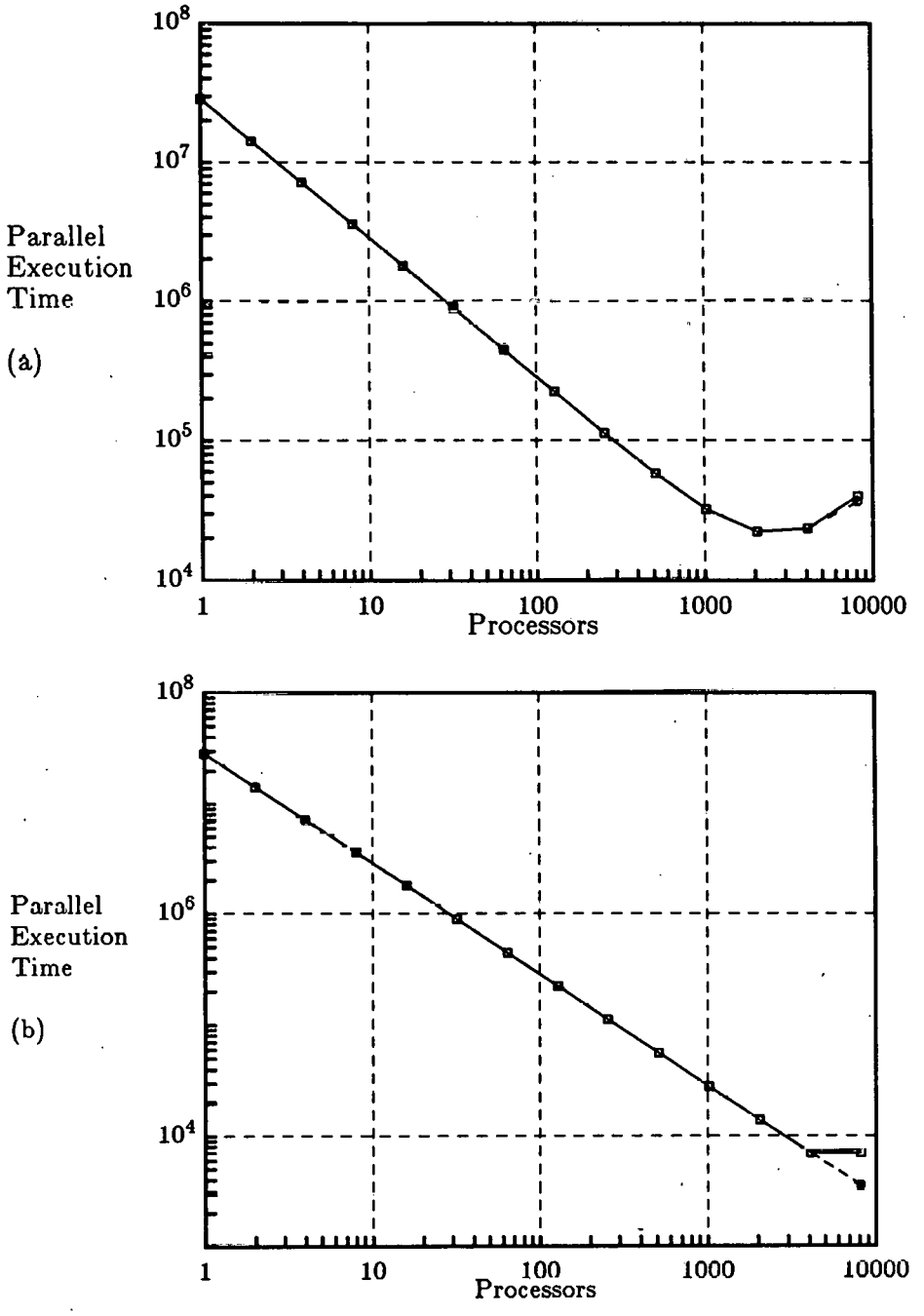


Figure 7.6. Execution times for (a) linear and (b) logarithmic overheads for N=4096 and B=7000.

Similarly (7.14) and (7.15) were used for the logarithmic overhead case.

Figure 7.3 illustrates the execution time for a DOALL with N=200 and B=8 under (a) linear overhead, and (b) under logarithmic overhead. Figures 7.4, 7.5, and 7.6 illustrate the same data for three different DOALLs. The solid lines plot the values of T_p, the actual parallel execution time. Dashed lines give the approximate execution time defined by the continuous functions. For these measurements a value of $\sigma = 4$ was used. The overhead constant although optimistically low, is not unrealistically small for (hypothetical) systems with fast synchronization hardware. In all cases we observe that as long as $p \leq N$, the difference between the values of the approximation function $T(p)$ and the actual parallel execution time T_p is negligible.

Looking at Figures 7.3a and 7.5a we observe that when the loop body is small, the associated overhead limits severly the number of processors that can be used on that loop. For these two cases for example, only 1/10 and 1/40 of the ideal speedup can be achieved. When B is large however the overhead has a less limiting impact on speedup. For the case of Figure 7.4a for example, 1/2 of the maximum speedup can be obtained in the presense of linear overhead. The same is true for Figure 7.6a. In all cases the logarithmic overhead had practically no significant negative impact on speedup.

7.6. Problems and Trade-offs in Dynamic Scheduling

For our discussion in this section we need to define more precisely what are the components of a program that we want to schedule, namely tasks and processes. We defined earlier the concept of basic block [AhUl77]. A straight line code (SLC) module is a piece of code that consists of a set of basic blocks with conditional statements embedded between basic blocks. The only other type of executable code are outermost loops. Outer loops can be arbitrarily complex and are treated separately. A task is an SLC module or an outermost loop. A task may consist of

several iterations or processes in which case we have a parallel task. Processes and straight line code modules are always executed on a single physical processor. Each process may have scalar or vector statements but never parallel components. Therefore the parallelism within a process is always of SIMD type. Parallelism in tasks is always of MIMD type.

In defining the above terms we considered the following aspects of Fortran programs and real parallel processor systems. Recall that our machine model is a general purpose parallel processor system with powerful processors whose utilization should be kept as high as possible. The processors may be multifunctional or pipelined and low level parallelism should be exclusively utilized within a processor. In other words we do not allow low level spreading across processors (where low level refers to statement or instruction level granularity). We therefore exclude from our definition of parallelism low level spreading and vector statements. Note that this assumption still allows for very long vector statements to be distributed across several processors. This can be done by breaking the vector statement into smaller segments at compile-time and creating an artificial parallel loop around the reduced size vector statement. This is known as strip mining.

If we adopt the above assumptions, there can be only two types of parallelism in a Fortran program: Parallelism due to high level spreading (vertical) and parallelism due to (possibly nested) parallel loops. Using the earlier results of this chapter we serialize (either at compile or at run-time) all those parallel loops whose parallel execution would potentially result in a slowdown, or no speedup due to the overhead involved. We can now consider the tradeoffs involved with scheduling these types of constructs.

All ready-to-execute tasks will be queued in a common pool Q that may be implemented to support parallel deletions and insertions. Each ready task in Q will be represented by a tem-

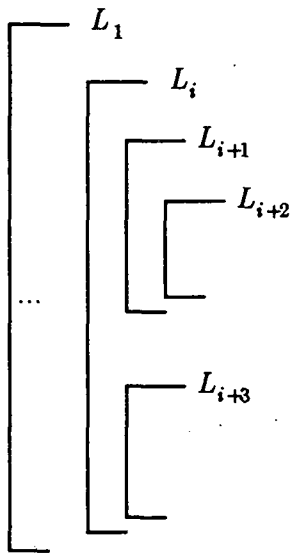


Figure 7.7. A multiply nested hybrid loop.

plate. Tasks in Q may be serial (serial loops or SLC) or parallel. When a processor dispatches a task from Q it may execute the entire task or part of it. The type of parallelism discussed above is explicitly represented in Q: All tasks in Q may be executed simultaneously (high level spreading). On the other hand parallel tasks in Q may be executed on several processors (loop parallelism).

Let us suppose that Q has parallel access capabilities and thus each processor can dispatch a task (or part of a task) from anywhere in Q. Therefore there is no complication with high level spreading. If Q for example contains serial tasks only, each incoming processor will dispatch a single task until all tasks in Q are dispatched. Some groups of processors may have to perform a barrier synchronization, but we are not concerned of how this can be done until later. The only problem left is how to dispatch a parallel task, that is, a parallel loop. Let us

look at the problems involved with distributing a parallel loop. We assume that each incoming processor dispatches a set of consecutive iterations (process) from a parallel loop. If we have a single level of parallelism the procedure is trivial. If however we have multiply nested parallel loops the dispatching process becomes more complicated. Consider the example loop of Figure 7.7 and let us consider the following scenarios.

All loops are DOALLs. Then if N_i is the number of iterations of L_i the first j loops for which $\prod_{i=1}^{j} N_i \geq p$ will be executed in parallel. All remaining loops will be serialized. Since we are not interested in the optimal allocation (since it is impossible in this case), our objective is to utilize enough parallelism to keep the processors busy. Because all loops are parallel the above selection will give us enough iterations (processes) to keep all p processors in the system busy. If there is no dependence from L_{i+1} to L_{i+3} and in addition $\prod_{j=1}^{i+2} N_j < p$, we should be able to perform high level spreading inside L_i by overlapping the execution of L_{i+1} and L_{i+3}. If all surrounding loops are DOALLs the compiler can distribute them around L_{i+3} and therefore L_{i+3} will appear in Q as a separate parallel task. High level spreading is then automatic. If one of the surrounding loops e.g., L_i is serial, loop distribution is not allowed and we should take provisions in our design to enforce or allow spreading within a parallel task. In this case overlapped execution of L_{i+1} and L_{i+3} can be achieved by using a barrier synchronization at the end of L_i. Incoming processors can dispatch iterations of L_{i+1} and L_{i+3} in any order, for the same value of the index of L_i, and as long as the barrier is cleared. Exactly the same could be done if the serial loop was L_j, for $1 \leq j < i$.

If there is a dependency from L_{i+1} to L_{i+3}, each processor that executes an instance of L_{i+1} for a given index value of L_i will also execute the corresponding instance of L_{i+3}. Therefore, since

dependences remain "within a processor" they are preserved automatically. No barrier synchronization between the two loops is needed in this case. The self-scheduling scheme proposed in [TaPe86] uses synchronization for each single loop in the construct. This approach involves a large overhead especially for small loops with many iterations. Moreover this method does not allow high level spreading within the scope of a loop. Later in this chapter we present a different self-scheduling scheme that uses far less synchronization and also allows for spreading inside a loop.

Our next goal is to design a dynamic scheduling scheme that is efficient, realistic, and involves as low an overhead as possible. Since scheduling is done in a non deterministic way in this case, we should design our scheme such that processors are scheduled automatically and select the "best" task to execute next by going through a simple and fast procedure. To achieve this we design our scheme around the following two objectives:

- Keep all processors as busy as possible.
- Run-time overhead should be kept minimal.

The following theorem clarifies our intention for using these two rules. Assuming zero overhead, and that we can keep all processors busy, we see from the next theorem that we can get very close to an optimal schedule by using any simple self-scheduling scheme.

Theorem 7.4 Let L be a DOALL loop with N iterations and a loop-body that takes a maximum of B units of time to execute. Then the execution time ω_L using any self-scheduling scheme is bounded by

$$\omega_L \leq \left\lceil \frac{N}{p} \right\rceil B. \tag{7.25}$$

Proof Let us assume that $N > p$. During self-scheduling iterations are scheduled on demand.

When a processor becomes free it dispatches a new iteration or a block of iterations. (Since the latter case can be reduced to the case where iterations are assigned one by one, we consider the former case.) In a DOALL there are no cross-iteration dependences. Therefore new processes are always available until L is completely dispatched. In other words there are no "gaps" in the execution profile.

Assume that all p processors start at time 1. Let t be the time the first of the p processors finishes completely (i.e., it finds an empty Q). Then no new iteration (process) can start execution at time greater than or equal to t. Therefore all other processors will complete before time $t + B$ and thus

$$\omega_L \leq t + B. \tag{7.26}$$

There is at least one (out of the p) processor that by time t has been assigned at most $\lceil N/p \rceil - 1$ iterations. This is true because otherwise (i.e., if each processor has been assigned at least $\lceil N/p \rceil$ iterations by time t) the total number of iterations x assigned up to time t would be $x \geq p\lceil N/p \rceil \geq N$ which is impossible (unless p divides N which again proves the theorem). It follows therefore that

$$t \leq \left(\left\lceil \frac{N}{p} \right\rceil - 1\right)B \quad \text{or} \quad t + B \leq \left\lceil \frac{N}{p} \right\rceil B \tag{7.27}$$

and if we substitute t in (7.26) we finally have

$$\omega_L \leq \left\lceil \frac{N}{p} \right\rceil B. \quad \blacksquare$$

Consider again a DOALL loop L with N iterations, or equivalently a set of N independent serial tasks. Let B and b be the execution times of the longest and shortest iterations of L respectively. Then if ω_o is the optimal schedule length of L on p processors and ω_L the schedule lenght of L on p processors under any dynamic (demand-driven) scheduling scheme (assuming no over-

head of any kind), we have the following.

Corollary 7.6 ω_L can never be B/b times worse than the optimal, that is

$$\frac{\omega_L}{\omega_o} \leq \frac{B}{b}.$$

Proof From the previous theorem it follows that

$$\left\lceil \frac{N}{p} \right\rceil b \leq \omega_o \leq \omega_L \leq \left\lceil \frac{N}{p} \right\rceil B, \quad \text{or}$$

$$\frac{\omega_L}{\omega_o} \leq \frac{\lceil N/p \rceil B}{\omega_o} \leq \frac{\lceil N/p \rceil B}{\lceil N/p \rceil b} = \frac{B}{b}.$$

For example if all iterations of L have equal execution times $\omega_L = \omega_o$, i.e., any demand-driven scheme is optimal, excluding again overhead. ■

Another upper bound which in general is closer than that of Corollary 7.6 is given by the following.

Corollary 7.7 $$\frac{\omega_L}{\omega_o} \leq 1 + \frac{p}{N} * \frac{B}{b}.$$

Proof Since t is the time the first processor completes execution on L (Theorem 7.4), it is obvious that

$$\omega_o \geq t \geq \left\lfloor \frac{N}{p} \right\rfloor b. \tag{7.28}$$

From the previous theorem we also have,

$$\omega_L = T_p \leq t + B. \tag{7.29}$$

From (7.28) and (7.29) it follows that

$$\frac{\omega_L}{\omega_o} \leq \frac{t + B}{\omega_o} \leq \frac{t + B}{t} = 1 + \frac{B}{t} \leq 1 + \frac{B}{\lfloor N/p \rfloor b} \leq 1 + \frac{p}{N} * \frac{B}{b} \quad (7.30)$$

which proves the corollary. ■

From Corollary 7.7 we observe that if N is very large compared to p, any dynamic scheduling heuristic converges to the optimal. The case of high level spreading is also included in the above theorem. In the case of general program graphs (DAG's) the above theorem holds true if we replace $\lfloor N/p \rfloor B$ by ω_o where ω_o is the length (execution time) of the optimal schedule for a particular program graph. Therefore in the worst case dynamic scheduling will result in a parallel execution time which is by at most t_g units of time longer than the optimal (where t_g is the execution time of the longest serial task in the graph). By using the compiler to guide dynamic scheduling, we can reduce t_g to be the execution time of a particular serial task. This is as good a performance as we can get given the nondeterministic nature of the problem. Since any dynamic scheduling algorithm is bounded by (7.25), we should use the simplest possible: for example processors pick random tasks from Q to execute next. This will work (theoretically) as well as any other more complicated scheduling procedure. The only problem (and in fact the dominant one in reality) is overhead. The overhead varies with different schemes. The execution time given by Theorem 7.4 can never be realized in real systems. Overhead is never zero but instead, it may be several times longer than the execution time of a task. When a task t_i is scheduled there is always an overhead factor o_i associated with it that prolongs its execution time. This overhead is paid by a processor each time that processor dispatches a new task. Therefore in reality, if a given program graph is scheduled on p processors with each processor executing an average of $\lceil N/p \rceil$ tasks, with an average task execution time of t_i the schedule length would be of the order $O(\lceil N/p \rceil (t_i + o_i))$ instead of $O(\lceil N/p \rceil t_i)$ as implied by Theorem 7.4, where o_i is the overhead factor. If $o_i > t_i$ which is often the case, then dynamic

scheduling may give an execution time several times longer than the best possible.

Our next design goal therefore is to implement dynamic scheduling such that o_i is minimized for each processor. Note also that if o_i was a constant it would be rather easy to optimize the schedule even at run-time. However the overhead o_i depends dynamically on several factors and may vary vastly from task to task and processor to processor. For example, o_i depends on the size of the task, the type of the task, the total number of processors that attempt a dispatch at the same moment, and so on. This becomes clear if we see what run-time activities "contribute" to o_i. Such activities for example include time spent in Q during dispatching. Obviously the fewer the processors that try to dispatch a task at a given moment, the shorter the time a processor will wait in Q. Since dynamic and in particular self-scheduling makes it by nature impossible to predict what task a processor will execute next, instruction and data fetching will have to be done after a processor dispatches a task. Although we discuss later ways of prefetching instructions and data, in a pure implementation of self-scheduling these activities cannot be overlapped with execution and they are attributed to o_i. Several processors may have to synchronize after executing a parallel task by performing a *join* operation which can be implemented with barrier synchronization. This overhead is also accounted for in o_i. If all the processors involved in a parallel task attempt a join simultaneously, the overhead for each processor would obviously be higher than if each processor performs the join at a different moment, or if the arrival of processors to the barrier follows some probabilistic distribution. Since a systematic analysis is impossible for the real problem, we should design a run-time scheduling scheme that, in a pure form, limits the overhead to be within reasonable bounds. Compiler generated information can then be used to enhance performance by forcing the processors to make more intelligent selection of tasks without extra overhead.

7.7. Self-Scheduling Through Implicit Coalescing

In this section we discuss a new self-scheduling scheme that requires minimal synchronization. In this case the concurrency control is fully distributed i.e., our machine model consists of p processors that operate independently and autonomously - no global control unit of any type exists. For simplicity we can assume that the machine has a fixed address space in global memory where the program graph (in some representation) is stored. There is also a common pool Q of ready-to-execute tasks. Q may support parallel task dispatch/insert operations. A possible implementation would be to maintain Q as a hash table. A processor then may access any entry of Q by computing the address using a hash function. In addition a processor may be forced to use the last hash address instead of recomputing a new one. This for instance may be useful when a processor executes part of a parallel loop. If the processor is forced in this case to dispatch another part of the same loop next we save instruction and fetches of read-only data. Another possible implementation of Q is a table where an idle processor dispatches its task from a random entry. There are several other ways of organizing Q but this is another subject on its own.

Another concern here is how to insert (enqueue) ready-to-execute tasks in Q, and how to determine whether a task is ready-to-execute. Since there is no global control unit in our system this should be the responsibility of the operating system or the program itself. The operating system however is activated only when it receives a specific request. This means that only periodic checking for ready-to-execute tasks is possible, clearly the least desirable approach. The best approach is to enqueue tasks in Q as soon as they become executable. This means that as soon as all their predecessors complete execution, one of the processors should be able to detect this event and "fire" the corresponding tasks. The best candidate for this, in terms of efficiency, is the processor that completes the barrier associated with each task (if any), or the processor

that dispatches the last part of a task. We can view this activity as service to the "community" of processors. In general the responsibility for this service should be evenly distributed among the p processors. Therefore for what follows we assume that Q always contains the ready-to-execute tasks and remains updated through program execution. The details of how this is done are of no concern to the following discussion. Again high level spreading is automatic. Therefore we present our self-scheduling scheme only in the context of arbitrarily nested hybrid loops.

Most of the schemes that have been proposed so far [GGKM83], [TaPe86], implement self-scheduling by making extensive use of synchronization instructions. For example in [TaPe86] a barrier synchronization is associated with each loop in the construct. In addition, all accesses to loop indeces are, by necessity, synchronized. Another common characteristic of these schemes is that they assign only one loop iteration to each incoming (free) processor. Our scheme differs in all aspects discussed above. Only one barrier per serial loop is used. Furthermore, independently of the nest pattern and the number of loops involved, we need synchronized access to only a single loop index. In contrast the above schemes need synchronized access to a number of indeces which is equal to the number of loops in the construct.

Self-scheduling can be achieved through loop coalescing. This compiler transformation was described in Chapter 4 and was used to enhance the performance of static-scheduling. The key characteristic of this transformation which is useful here, is its ability to express all indeces in a loop nest as a function of a single index. This makes it clear why synchronized access to each loop index is wasteful. We can always use a single index. If the loop bounds are known at run-time just before we enter the loop, we may decide exactly how many iterations each processor will receive. Thus when a processor accesses the single loop index to dispatch a range of consecutive iterations it goes through a single synchronization point. Since the range of iterations is determined before-hand, each processor will dispatch all the work it is responsible for, the very

first time it accesses the corresponding loop index. Therefore only a total of p synchronization instructions will be executed. For a matter of comparison, in the schemes mentioned above each processor executes a synchronization instruction for each loop in the nest, and each time it dispatches a new iteration. In a nested loop that consists of m separate loops we would then have a total of $m\prod_{i=1}^{m} N_i$ synchronization instructions that will execute before the loop completes. The difference between p and $m\prod_{i=1}^{m} N_i$ can obviously be tremendous. In theoretical terms we can thus state that the scheme in [TaPe86] or [GGKM83] for example, involves an overhead which is unbounded on p.

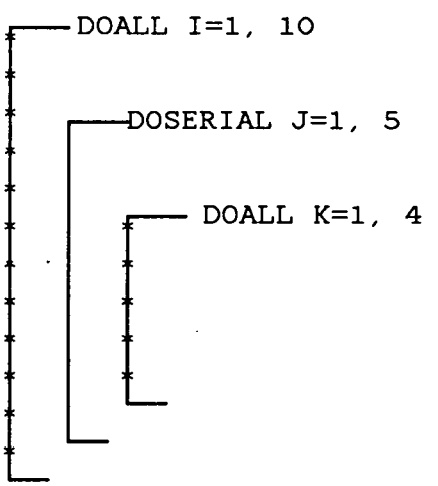


Figure 7.8. Example loop for the application of GSS.

7.7.1. The Guided Self-Scheduling (GSS(k)) Algorithm

In this section we present a simple, yet powerful algorithm for dynamic scheduling. The idea is to implement *Guided Self-Scheduling* with bound k, (GSS(k)) by "guiding" the processors on the amount of work they choose. The *bound* is defined to be the minimum number of loop iterations assigned to a given processor by GSS. The algorithm is discussed below in great detail and is summarized for $k=1$ in Figure 7.10. First we present the case of $k=1$, GSS(1) or GSS for short, and later discuss the general case for $k > 1$. The GSS algorithm achieves optimal execution times in most cases. Let us describe in more detail how self-scheduling through implicit loop coalescing works. For the beginning assume that we have a perfectly (one-way) nested loop $L = (N_1, ..., N_m)$. As discussed in Chapter 4, loop coalescing coalesces all m loops into a single loop $L' = (N = \prod_{i=1}^{m} N_i)$ through a transformation f that maps the indeces I_i, $(i = 1, 2,..., m)$ of the original loop L to index I of L' such that $I_i = f_i(I)$ $(i = 1, 2,..., m)$. This index transformation is universal, i.e., it is the same for all loops, perfectly nested or not. Therefore each processor can compute locally f_i for a given I. Better yet, each processor can compute locally a range of values $f(x:y)$ for a range of $x \leq I \leq y$. This mapping f as defined in (4.4) (Chapter 4) may be implemented in microcode, or a fast hardware device may be used inside each processor to realize f. The global index I is then kept in the shared memory as a shared variable. Each processor accesses I in a synchronized way and dispatches the next set of consecutive iterations of L along with a pointer to its code. Then inside each processor, mappings f_i are used to compute the corresponding range for each index I_i of the original loop. After the index ranges are computed for each processor, execution proceeds in the normal mode. In case all loops in L are parallel no processor will ever go back to dispatch another range of iterations of I. This is obviously the minimum possible

amount of synchronization that is needed with any self-scheduling scheme.

The process is more complicated with self-scheduling of hybrid loops. Let us look at the case of hybrid loops that consist of DOALLs and DOSERIAL loops, and in particular consider the example of Figure 7.8. In the example the innermost and outermost loops are DOALLs and the second is a serial loop. Let us denote this loop with $L = (N_1, N_2, N_3) = (10, 5, 4)$. We have a total of $N = 200$ iterations. On a machine with an unlimited number of processors (200 in this case) each processor would execute 5 iterations of L, and this is the best possible that we can achieve. On a system with p processors self-scheduling should be done such that iterations of L are evenly distributed among the p processors (assuming an equal execution time for all iterations). The presence of the serial loop in L however limits our ability to do this. It is profound that the approach of assigning consecutive iterations of I to each processor would fail here. (This is true because after coalescing we have a single iteration space and assignments are done in blocks of consecutive iterations.) At most 4 successive iterations may be assigned at once. If all 4 are given to the same processor, the loop is executed serially. If each processor receives one iteration on the other hand, we can use only up to 4 processors.

This problem can be eliminated by permuting the indeces of the original loop, or equivalently, by applying implicit loop interchange [Wolf82]. Our goal is to permute the indeces so that the longest possible set of parallel iterations corresponds to successive values of the index I of L'. This can be done by permuting the indeces I and J so that the serial loop becomes the outermost loop or by permuting J and K so that the serial becomes the innermost loop - which would violate dependences in this case. In general a serial loop can be interchanged with any DOALL that surrounds it, but it may never be interchanged by a loop surrounded by it. Therefore in the case of our example we implicitly interchange loops I and J.

The interchange can be implemented trivially using implicit coalescing as follows. The mappings of I and J are permuted such that I is defined by the mapping of J and vice versa. No physical loop interchange takes place (neither physical coalescing). More specifically, if I_c is the global index of the coalesced loop for the example loop of Figure 7.8, then the original indeces I, J and K are mapped to I_c as follows:

$$\begin{aligned} I &= \left\lceil \frac{I_c}{20} \right\rceil - 10 \left\lfloor \frac{I_c - 1}{200} \right\rfloor \\ J &= \left\lceil \frac{I_c}{4} \right\rceil - 5 \left\lfloor \frac{I_c - 1}{20} \right\rfloor \\ K &= I_c - 4 \left\lfloor \frac{I_c - 1}{4} \right\rfloor . \end{aligned} \tag{7.31a}$$

After implicit loop coalescing the mappings are:

$$\begin{aligned} I &= \left\lceil \frac{I_c}{4} \right\rceil - 10 \left\lfloor \frac{I_c - 1}{40} \right\rfloor \\ J &= \left\lceil \frac{I_c}{40} \right\rceil - 5 \left\lfloor \frac{I_c - 1}{200} \right\rfloor \\ K &= I_c - \left\lfloor \frac{I_c - 1}{4} \right\rfloor . \end{aligned} \tag{7.31b}$$

The result is that the first 40 successive values of I_c correspond now to 40 parallel iterations (instead of 4 iterations previously). Therefore up to 40 processors can be used in parallel. Extra synchronization is still needed however. As mentioned earlier in this section, each serial loop in L needs a barrier synchronization to enforce its seriality. The following lemma tells us when it is legal to apply loop interchange in order to maximize the number of consecutive parallel iterations.

Proposition 7.1. In a hybrid perfectly nested loop, any DOALL can be interchanged with any serial or DOACR loop that is in a higher nest level. This loop interchange can be applied repeatedly and independently for any pair of (DOALL, DOSERIAL / DOACR) loops.

Proof The proof is trivial for the case of two loops. The general case follows by induction on the number of loops interchanged. ■

The only case that remains to be discussed is nonperfectly (multi-way) nested loops. This is identical to the one-way nested loop case, unless one of the following two conditions is met. 1) Loops at the same nest-level have different loop bounds. 2) High level spreading should be applied with loops at the same nest-level. In the first case if k loops N_{i+1}, N_{i+2}, . . . , N_{i+k} happen to be at the i-th level, the global index I_c is computed with a number of N_i iterations for the i-th level, which is given by

$$N_i = \max_{1 \le j \le k} \{N_{i+j}\}.$$

Then during execution, loop N_{i+j} at the i-th level will have $N_i - N_{i+j}$ null iterations (which are not actually computed). Therefore some of the processors execute only part of the code at level i. This corresponds to computing "slices" of each loop on the same processor. Consider for example the loop of Figure 7.7. If L_{i+1} and L_{i+3} are independent, then only one mapping function $f_{i+1}(^*)$ can be used for both L_{i+1} and L_{i+3}. Thus slices of the two loops corresponding to the same index values will be assigned to each idle processor. In general if loops at the same nest level are independent, outer loops can be distributed around them and each loop is considered separately (i.e., we coalesce each of them and consider them as separate tasks). When there are dependences among loops, either loop distribution or barrier synchronization can be used as mentioned above. If for example there is a dependence from L_{i+1} to L_{i+3}, a barrier can be inserted between the two loops to insure completion of L_{i+1} before L_{i+3} starts executing.

If high level spreading is to be applied, then implicit loop coalescing and a global index I_c will be computed for each loop that is spread. Consider Figure 7.7 of the previous section. If loops L_{i+1} and L_{i+3} are to be overlapped, two implicit coalescings will be performed for loops $L_1, \ldots, L_i, L_{i+1}$ and $L_1, \ldots, L_i, L_{i+3}$ that will produce two different global indeces I_c^1 and I_c^2 respectively. A separate task for each of the I_c^1 and I_c^2 will then be created and queued in Q.

So far we saw how GSS coalesces the loops and assigns blocks of iterations to incoming (idle) processors. We have not mentioned however how the algorithm decides the number of iterations to be assigned to each idle processor. The schemes that have been proposed so far [KrWe85], [TaPe86] assign a single iteration at a time. This approach involves a tremendous amount of overhead since several critical regions must be accessed each time a single iteration is dispatched. The GSS algorithm follows another approach by assigning several (blocks of) iterations to each processor. The size of each block varies and is determined by using a simple but powerful rule that is described below. Before we describe how block sizes are computed let us state our constraints.

Suppose that a parallel loop L (e.g., a DOALL) is to be executed on p processors. We assume that each of the p processors starts executing some iteration(s) of L at different times (i.e., not all p processors start computing L simultaneously). This is clearly a valid and practical assumption. If L for example is not the first loop in the program, the processors will be busy executing other parts of the program before they start on L. Therefore they will start executing L at different times which may vary significantly. (Of course one could force all p processors to start on L at the same time, by enforcing a join (or barrier) operation before L; this would clearly be very inefficient.) Given now the assumption that the p processors will start executing L at arbitrary times , our constraint is to dispatch a block of consecutive iterations of L to each

incoming processor, such that all processors terminate at approximately the same time. This is a very desirable property. If L for example is nested inside a serial loop L_s, then a barrier synchronization must be performed each time L completes (i.e., for each iteration of L_s). If the processors working on L do not terminate at the same time, a very significant amount of idle processor time (overhead) may be accumulated by the time L_s completes.

Actually the best possible solution is that which guarantees that all p processors will terminate with at most B units of time difference from each other; where B is the execution time of the loop body of L. This goal can be achieved if blocks of iterations are assigned to idle processors following the next principle. An incoming processor P_x will dispatch a number of iterations x considering that the remaining $p-1$ processors will also be scheduled at this (same) time. In other words P_x should leave enough iterations to keep the remaining $p-1$ processors busy (in case they all decide to start simultaneously) while it will be executing its x iterations. If N is the total number of iterations, this can be easily done as follows:

$$\text{1st processor receives } \left\lceil \frac{N}{p} \right\rceil \text{iterations,}$$

$$\text{2nd processor receives } \left\lceil \frac{N - \lceil N/p \rceil}{p} \right\rceil \text{iterations,}$$

$$\text{3d processor receives } \left\lceil \frac{N - \lceil (N - \lceil N/p \rceil)/p \rceil}{p} \right\rceil \text{iterations,}$$

and so on. Since GSS coalesces loops, there will be a single index $I_c = 1...N$, from which idle processors will dispatch blocks of iterations. Therefore the assignment of iteration blocks is done by having each idle processor perform the following operations:

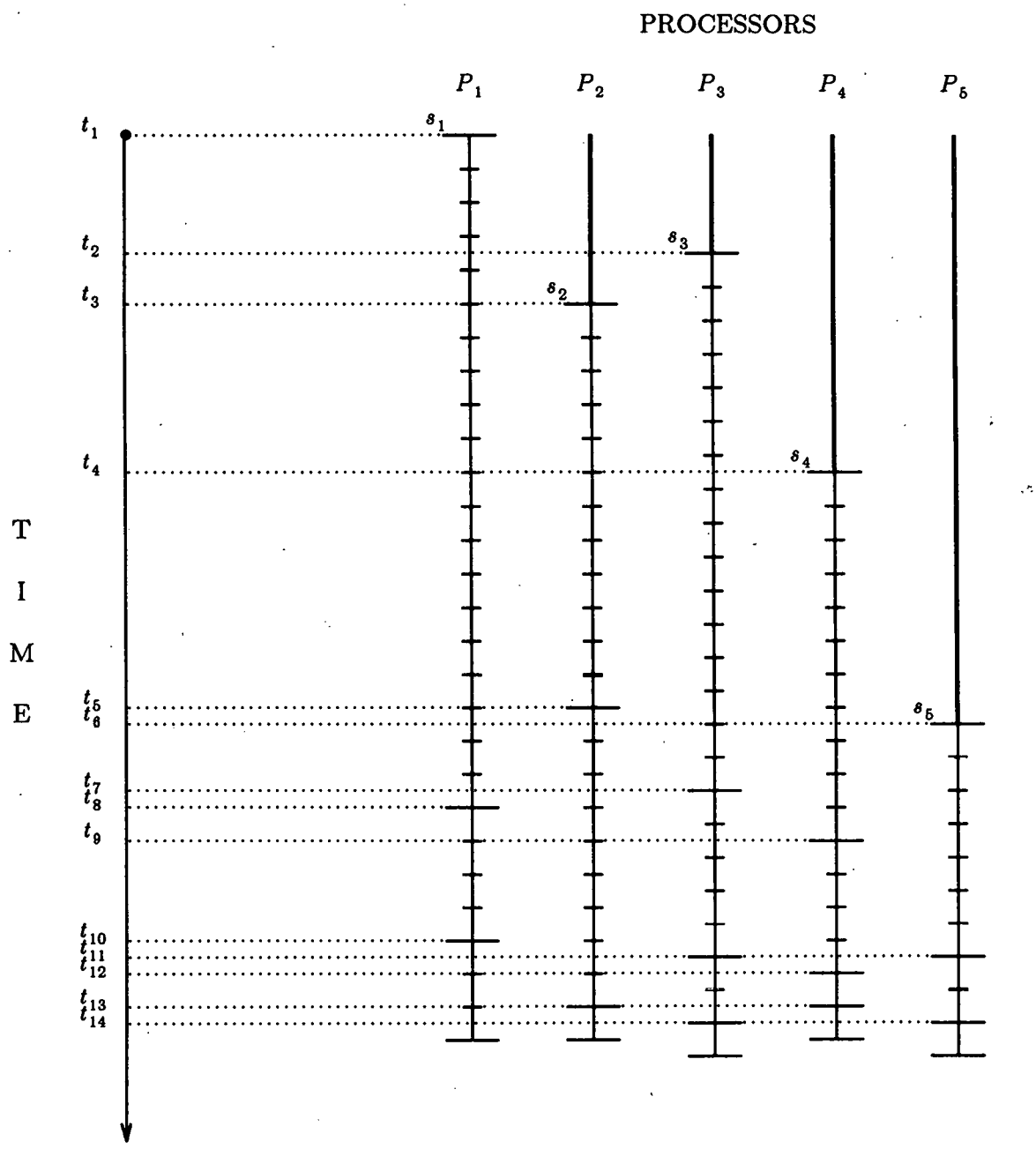


Figure 7.9. An example of the application of the GSS algorithm for $N=100$, $p=5$.

$$w = I_c; \quad x = \left\lceil \frac{N - I_c}{p} \right\rceil; \quad I_c \leftarrow I_c + x.$$

The range of iterations assigned to that processor is then given by $[w,..., w + x-1]$. The same operations can also be described for the i-th idle processor as follows:

$$x_i = \left\lceil \frac{R_i}{p} \right\rceil; \quad R_{i+1} \leftarrow R_i - x_i, \tag{7.32}$$

and the range of iterations for the i-th processor is given by $[x_{i-1} + 1,..., x_i]$, where $x_o = 1$ and $R_1 = N$.

Time	No. of unused iterations (I)	Next processor to be scheduled	No. of iterations assigned to this processor
t_1	100	P_1	20
t_2	80	P_3	16
t_3	64	P_2	12
t_4	52	P_4	11
t_5	41	P_2	9
t_6	32	P_5	7
t_7	25	P_3	5
t_8	20	P_1	4
t_9	16	P_4	4
t_{10}	12	P_1	3
t_{11}	9	P_3	2
t_{12}	7	P_5	2
t_{13}	5	P_4	1
t_{14}	4	P_1	1
t_{15}	3	P_2	1
t_{16}	2	P_3	1
t_{17}	1	P_5	1
			TOTAL= 100

Table 7.1. The detailed scheduling events of the example of Figure 7.9 ordered by time.

As an example, consider the case of a DOALL L with N=100 iterations that executes on five processors. All five processors start on L at different times. Each idle processor is assigned a block of consecutive iterations using the rule described above. The resulting execution profile is shown in Figure 7.9. Even though the results presented in this section hold for the general case where different iterations of the same loop have different execution times, for this example we assume that all 100 iterations have equal execution times. Each line segment in Figure 7.9 represents the execution time of a loop iteration. The thick lines represent the execution of previous (unrelated to L) tasks on processors P_2, P_3, P_4, and P_5. The wider horizontal line segments mark the time when iteration blocks are actually dispatched by idle processors. For example, at time t_1 processor P_1 dispatches $\lceil 100/5 \rceil$=20 iterations. The next processor to become available is P_3 which at time t_2 dispatches $\lceil (100\text{-}20)/5 \rceil$=16 iterations. Processor P_1 will receive its next assignment at time t_8. The detailed assignment of iterations to processors for this example is shown in Table 7.1. The events in the table are ordered by virtual time. We observe that although the five processors started executing L at different times, they all terminated within B units of time difference from each other. In general if p processors are assigned to a (coalesced) DOALL with N iterations using the above scheme, we have the following.

Lemma 7.1 Each of the last p-1 processors to be scheduled under the GSS algorithm is assigned exactly one iteration of L.

Proof Let r be the number of remaining iterations before the last p-1 processors are scheduled. Suppose that the statement of the lemma is not true. Then at least the first (of the last p-1 processors to be scheduled) is assigned 2 or more iterations. Equivalently $\lceil r/p \rceil \geq 2$, or $r > p$. We distinguish here two cases:

Case 1: $r = p + 1$.

In this case the $(p-1)$-th processor receives 2 iterations and there are $p-1$ iterations left. However each of the remaining $p-2$ processors will receive exactly one iteration (since the assignment is now computed from $\lceil x/p \rceil$ where $x \leq p-1$) and we are thus left with 1 unassigned iteration. This contradicts the initial hypothesis that the $p-1$ processors are the last to compute L.

Case 2: $r > p + 1$.

Using the same reasoning as in Case 1, we conclude that there are at least two unassigned iterations in this case, which again contradicts the initial hypothesis. The statement of the lemma is therefore true. ■

Theorem 7.5 Independently of the initial configuration (start-up time) of the p processors that are scheduled under GSS, all processors finish executing L within B units of time difference from each other.

Proof We will prove the theorem for the case where all p processors start executing L simultaneously. The proof for the general case is similar. By Lemma 7.1, at least the last $p - 1$, and at most the last p assignments will involve single iterations. Let us consider the latter case. The are two possible scenarios during the scheduling of L under GSS. In the first case each of the last p iterations is assigned to a different processor. Then by virtue of GSS it is easy to see that if t_i, t_j are the termination times for processors p_i, p_j, $i, j \in [1..p]$ respectively, we have $|t_i - t_j| < B$.

The second case is when the last p iterations of L are assigned to at most $p - 1$ different processors. Let p^x denote a processor whose last assignment was an iteration block of size x, and p^1 a processor that is assigned one or more of the last p iterations. Using the same argument as above we can show that all processors that received one of the last p iterations, finish within B units of time apart from each other. It remains to show that any p^x and any p^1 terminate within

B units of time from each other. We consider the case for $x=2$. The general case is similar. We will prove that for any p^x and any p^1, $|p^x - p^1| \leq B$.

Case 1: $p^x > p^1$, and suppose that $p^x - p^1 > B$ or equivalently, $p^x - 2B > p^1 - B$. The last inequality implies that p^1 was assigned a single iteration before the iteration block of size $x=2$ was assigned to p^x. Clearly this contradicts the basic steps of the GSS algorithm. Therefore $p^x - p^1 \leq B$.

Case 2: $p^x < p^1$ and suppose $p^1 - p^x < B$ or, $p^1 - B > p^x$. But the last inequality can never be true since p^x would have been assigned the last iteration instead of p^1. Therefore $p^1 - p^x \leq B$ and thus the statement of the theorem is true.

Note that if the p processors start executing L at different times $t_1 \leq t_2 \leq .. \leq t_p$, the theorem still holds true under the following condition:

$$N > \frac{1}{B} \sum_{i=1}^{p} (t_p - t_i). \qquad \blacksquare$$

In reality B varies for different iterations and $B \in \{b_1, b_2, ..., b_k\}$, where b_i, $(i=1,..., k)$ are all possible values of B. Suppose that B can assume any of its possible values with the same probability, i.e., $P[B=b_i]=1/k$, $(i=1, 2,..., k)$. Then Lemma 7.1 and Theorem 7.5 are still valid. Under the above assumptions we also have the following.

Theorem 7.6 The GSS algorithm obtains the optimal schedule under any initial processor configuration. Because of this optimal schedule, GSS also uses the minimum possible number of synchronization points.

By synchronization points we mean the number of times processors enter critical regions (i.e., loop indexing). An implementation of GSS can be done so that when q (out of the p) processors become simultaneously available at time t, the first $q-1$ receive $\lceil N_t / p \rceil$ iterations and

the q-th processor receives $\min(\lceil (N_t - (q-1)\lceil N_t/p \rceil)/p \rceil, \lceil N_t / p \rceil)$ iterations, where N_t is the number of unassinged iterations at time t.

Theorem 7.7 The number of synchronization points required by GSS is p in the best case, and $O(pH_{\lceil N/p \rceil})$ in the worst case, where H_n denotes the n-th harmonic number and $H_n \approx \ln(n) + \gamma + 1/2n$ (γ is Euler's constant).

Proof The best case is obvious from the above discussion. In general it is clear that the number of iterations assigned to each processor will be (possibly multiple) occurrences of the integers

$$\left\lceil \frac{N}{p} \right\rceil, \quad \left\lceil \frac{N}{p} \right\rceil - 1, \quad \left\lceil \frac{N}{p} \right\rceil - 2, \ldots, 1$$

in this order. Obviously there will be at least $p-1$ and at most p assignments of exactly one iteration. It can also be observed that the number of different assignments of iteration blocks of size $\lceil N/p \rceil - k$, $(k = 1, 2, \ldots, \lceil N/p \rceil - 2)$ depends on the relative values of p and $\lceil N/p \rceil - k$. More precisely, we can have at most

$$\left\lceil \frac{p}{\lceil N/p \rceil - k} \right\rceil, \quad (k = 1, 2, \ldots, \lceil N/p \rceil - 2)$$

different assignments of iteration blocks of size $\lceil N/p \rceil - k$. Therefore the total number of different assignments and thus the total number σ of synchronization points in the worst case is given by

$$\sigma \leq p + \sum_{i=2}^{\lceil N/p \rceil} \left\lceil \frac{p}{i} \right\rceil = \sum_{i=1}^{\lceil N/p \rceil} \left\lceil \frac{p}{i} \right\rceil.$$

For computing the order of magnitude we can ignore the ceiling and finally have

$$\sigma \approx \sum_{i=1}^{\lceil N/p \rceil} \frac{p}{i} = p \sum_{i=1}^{\lceil N/p \rceil} \frac{1}{i} = pH_{\lceil N/p \rceil}.$$

Therefore the number of synchronization points in the worst case is $\sigma = O(pH_{\lceil N/p \rceil})$. ■

The GSS Algorithm

Input An arbitrarily nested loop L, and p processors.

Output The optimal dynamic schedule of L on the p processors. The schedule is reproducible if the execution time of the loop bodies and the initial processor configuration (of the p processors) are known.

- Distribute the loops in L wherever possible.
- For each ordered pair of (DOALL, DOSERIAL/DOACR) loops, (where the DOALL is the outer loop) perform loop interchange.
- Apply implicit loop coalescing, and let I_c be the index of the coalesced iteration space.
- For each index i_k of the original loop define the index mapping,

$$i_k = f_{i_k}(I_c)$$

- If N_t is the number of remaining iterations at time t, then set $N_t = N$ and for each idle processor do.

REPEAT

- Each idle processor (scheduled at time t) receives

$$x = \left\lceil \frac{N_t}{p} \right\rceil$$

iterations.

- $N_t = N_t - x$
- The range of each original loop index for that processor is given by

$$i_k \in [f_{i_k}(I_c), \ldots, f_{i_k}(I_c) + \lceil N_k / p \rceil - 1]$$

UNTIL ($N_t = 0$)

Figure 7.10. The GSS Algorithm

Note that if barriers are used, GSS can coalesce all loops serial and parallel. The transformation as presented in Chapter 5 coalesces together only DOALL loops and leaves serial loops unchanged. This does not have to be the case in GSS however. Consider for example a DOALL loop with $\prod_{i=1}^{m} N_i$ iterations which is the result of coalescing m DOALLs. Suppose now that this DOALL is nested inside a serial loop with M iterations. GSS works fine on this doubly nested loop but it still must access two shared variables (loop indeces) for each assignment. The other alternative is to implicitly coalesce the serial and parallel loops into a single *block-parallel* loop or BDOALL with MN iterations. To do this a barrier synchronization must be executed every N iterations. If $I_c = 1...MN$, the number of remaining iterations R_i (in 7.32) still assumes an initial value N. The difference here is that each time $I_c \bmod N = 0$ a barrier synchronization is executed and R is reinitialized to N. This happens M times before the entire loop completes execution.

In the last section we discuss a centralized scheduling approach that supports prefetching of instructions and data.

7.7.2. Further Reduction of Synchronization Operations

Another interesting feature of the GSS algorithm is that it can be tuned to further reduce the number of synchronization operations that are required during scheduling. As mentioned above, the last $p - 1$ allocations performed by GSS, assigned exactly one iteration to each processor. The synchronization overhead involved in these $p - 1$ allocations may still be very high, especially when p is very large and the loop body is small.

We shall see now how to eliminate the last $p - 1$ assignments of single iterations of GSS. In fact we can eliminate all assignments of iteration blocks of size k ($< \lceil N/p \rceil$) or less. Let us discuss first the problem of eliminating assignments of single iterations from GSS. We show how

this can be done by means of an example. Consider the application of GSS to a DOALL with $N=14$ iterations on $p=4$ processors. The assignment of iterations to processors is shown below in detail:

1st assignment gives $\lceil 14 / 4 \rceil = 4$ iterations
2nd " " " " " " $\lceil 10 / 4 \rceil = 3$ " " " "
3rd " " " " " " $\lceil 7 / 4 \rceil = 2$ " " " "
4th " " " " " " $\lceil 5 / 4 \rceil = 2$ " " " "
5th " " " " " " $\lceil 3 / 4 \rceil = 1$ " " " "
6th " " " " " " $\lceil 2 / 4 \rceil = 1$ " " " "
7th " " " " " " $\lceil 1 / 4 \rceil = 1$ " " " ".

The seven successive assignments were done with iteration blocks of size 4, 3, 2, 2, 1, 1, 1. In this case the single iteration assignments account for almost half of the total assignments. We can eliminate the single iteration assignments by increasing the block size of the first $p - 1$ assignments by 1. The successive assignments in that case would be 5, 4, 3, and 2. Therefore the total number of scheduling decisions (and thus synchronization operations) is reduced by $p - 1$. This technique can be applied automatically by setting $R_1 = N + p$ in (7.32). Thus the first assignment will dispatch $x_1 = \lceil (N+p)/p \rceil$ iterations. GSS is applied in precisely the same way. However now it terminates not when the iterations are exhausted, but when for some i, $x_i < 2$. For the above example the application of GSS will generate the following assignments ($R_1 = \lceil (N+p)/p \rceil$).

1st assignment gives $\lceil 18 / 4 \rceil = 5$ iterations
2nd " " " " " " $\lceil 13 / 4 \rceil = 4$ " " " "
3rd " " " " " " $\lceil 9 / 4 \rceil = 3$ " " " "
4th " " " " " " $\lceil 6 / 4 \rceil = 2$ " " " ".

When the ratio N/p is rather small, GSS(k) for $k=2$ may result in considerable savings. There is still a drawback however, since the rule of making all the assignments of iteration blocks of size two or more is not always accurate. Consider again the previous example but now let $N=15$. The assignments generated by GSS(2) will now be:

$$\lceil 19 / 4 \rceil = 5 \quad \lceil 14 / 4 \rceil = 4 \quad \lceil 10 / 4 \rceil = 3 \quad \lceil 7 / 4 \rceil = 2 \quad \lceil 5 / 4 \rceil = 2.$$

But $5 + 4 + 3 + 2 + 2 = 16 > N = 15$, i.e., the number of iterations assigned by GSS(2) is more than the iterations of the loop. Fortunately the number of superflows iterations in such cases cannot be more than one, and the termination problem can be easily corrected. The solution is given by the following theorem.

Theorem 7.8 Let k be the step in (7.32) such that $x_k = 2$ and $x_{k+1} = 1$. If $R_{k+1} = p$ then

$$\sum_{i=1}^{k} x_i = N$$

else, if $R_{k+1} = p - 1$ then

$$1 + \sum_{i=1}^{k-1} x_i = N.$$

Proof: The algorithm starts with a total of $N + p$ iterations, and it must assign a total of N iterations in blocks of size ranging from $\lceil (N+p)/p \rceil$ to 2. Since (for $p \geq 2$) at least one iteration block will be of size 2, and all assignments of iteration blocks of size 2 must be performed, it follows that the last assignment of GSS(2) will involve $R_k = p + 1$ or $R_k = p + 2$. In the latter case the last assignment will dispatch 2 iterations and the algorithm will terminate assigning therefore a total of $N + p - R_{k+1} = N$ iterations. If $R_k = p + 1$, the last assignment will also dispatch 2 iterations. In that case however the total number of iterations assigned will be $N + p - (p - 1) = N + 1$. Thus $1 + \sum_{i=1}^{k-1} x_i = N$. ■

Theorem 7.8 supplies the test for detecting and correcting superflows assignments. The assignment and termination condition for GSS(2) is now given by

$$x_i = \left\lceil \frac{R_i}{p} \right\rceil; \quad R_{i+1} \leftarrow R_i - x_i \tag{7.33}$$

```
if (R_{i+1} ≤ p) then
    { stop;
      if (R_{i+1} < p) then x_i = 1 }
```

Using (7.33) now, the last assignment of GSS(2) for the last example will dispatch a single iteration. The same process can be applied to derive GSS(k) for any $2 < k < \lceil N/p \rceil$. The best value of k is machine and application dependent.

```
DOALL 1 I = 1, N
      . . .
      . . .
      . . .
ENDOALL
```

(a)

```
DOSERIAL 1 I = 1, K
   DOALL 2 J = S(I), S(I)+B(I)
      . . .
      . . .
      . . .
   ENDOALL
ENDOSERIAL
```

(b)

Figure 7.11. Example of the application of GSS at the program level.

It should be emphasized that the GSS scheme can be implemented in hardware, it can be incorporated in the compiler, or it can be explicitly coded by the programmer. In the latter case the programmer may compute the iteration block size for each assignment, and force the assignment of such blocks by coding the corresponding loop appropriately. Consider for example the loop of Figure 7.11(a). If array B holds the block size and S holds the starting iteration for each assignment, the loop of Figure 7.11(a) can be coded as in Figure 7.11(b). Assuming that self-scheduling (SS) is implemented in the target machine, the above loop will be executed as if GSS was supported by the machine (with some additional overhead involved with the manipulation of the bookkeeping arrays).

7.7.3. Simulation Results

A simulator was implemented to study the performance of self-scheduling (SS) and GSS (GSS(1)). The simulator was designed to accept program traces generated by Parafrase, and it can be extended easily to implement other scheduling strategies. The experiments conducted for this work however, used four representative loops which are shown in Figure 7.12.

7.7.3.1. The Simulator

The simulator input consists of a set of tuples, where each tuple represents a single loop or a block of straight-line code. Each tuple includes information such as number of iterations, execution time of basic blocks inside the loop, branching frequencies for the branches of each conditional statement inside a loop, dependence information, type of loop etc. In the presence of conditional statements the conditions are "evaluated" separately for each iteration of the loop and the appropriate branch is selected. The user supplies the expected frequency with which each branch is selected. Otherwise the simulator considers each branch equally probable. For this purpose a random number generator is used with a period of $2^{31} - 1$ [Knut81], [Koba81]. Random

numbers are generated using uniform distribution and are normalized (in [0..1]). For each conditional statement in the loop, the interval [0..1] is partitioned into a number of subintervals equal to the number of branches in that statement. The size of each subinterval is proportional to the expected frequency of that branch. For each iteration of the loop, a random number is generated and the subinterval to which it belongs is determined. Then the branch corresponding to that subinterval is taken.

The execution of arbitrary loops on systems with 2 to 4096 processors can be simulated. Processors can start on a loop at random times. The simulator takes also into account overhead incurred with operations on shared variables. For our purposes shared variables are considered to be only loop indeces. Although the current version of the simulator assumes a fixed execution time for each BAS, it can be easily extended to operate on a program trace and take into account random delays (due to network contention in shared memory systems). For each memory access, a random delay may be computed to fall within given upper and lower bounds. These bounds may be readjusted each time the number of processors (and thus the number of stages of the network) grows.

7.7.3.2. Experiments

The four loops $L1$, $L2$, $L3$, and $L4$ of Figure 7.12 were used to conduct the experiments for this work. These loops are representative of those found in production numerical software. Serial and parallel loops are specified by the programmer, or are created by a restructuring compiler (e.g., Parafrase). The loops of Figure 7.12 cover most cases since they include loops that are i) all parallel and perfectly nested ($L1$), ii) hybrid and perfectly nested ($L3$), iii) all parallel and non-perfectly nested ($L2$), iv) hybrid nonperfectly nested ($L4$), v) and finally one-way ($L2$), and multi-way nested ($L4$). The arrows in $L4$ indicate flow dependences between adjacent loops. The numbers enclosed in angle brackets give the execution times of BASs in the corresponding

```
                DOALL 1 I1 = 1, 100
                  DOALL 2 I2 = 1, 50
                    DOALL 3 I3 = 1, 4
L1:                       {20}
                          [if C then {10}]
                    ENDOALL
                  ENDOALL
                ENDOALL
```

(a)

```
                DOALL 1 I1 = 1, 50
                   {5}
                   [if C then {10}]
                   DOALL 2 I2 = 1, 40
                      {5}
                      DOALL 3 I3 = 1, 4
L2:                      {10}
                         [if C then {20}]
                      ENDOALL
                   ENDOALL
                ENDOALL
```

(b)

```
                DOSERIAL 1 I1 = 1, 40
                   DOALL 2 I2 = 1, 500
L3:                   {100}
                      [if C then {50}]
                   ENDOALL
                ENDOSERIAL
```

(c)

CONTINUED (Figure 7.12)

(CONTINUED)

```
DOSERIAL 1 I1 = 1, 50

   DOALL 2 I2 = 1, 10
     DOALL 3 I3 = 1, 10
       DOALL 4 I4 = 1, 4
         {10}
         [if C then {50}]
       ENDOALL
     ENDOALL
   ENDOALL

            |
            V

   DOALL 5 I5 = 1, 100
L4:  {50}
     DOALL 6 I6 = 1, 5
       {100}
       [if C then {30}]
     ENDOALL
   ENDOALL

            |
            V

   DOALL 7 I7 = 1, 20
     DOALL 8 I8 = 1, 4
       {30}
     ENDOALL
   ENDOALL

ENDOSERIAL
```

(d)

Figure 7.12. Loops $L1$, $L2$, $L3$, and $L4$ used for the experiments.

positions.

Two sets of experiments were conducted, E_1 and E_2. The first set used the four loops of Figure 7.12 ignoring the conditional statements which are enclosed in square brackets. Therefore for E_1 all iterations of a particular loop had equal execution times. For E_2 the conditional statements were taken into account as well. Thus in E_2 different iterations of a given loop had different execution times. The next step will be to consider loops with multiple and nested conditionals which were not included in these experiments.

Earlier in this chapter we discussed extensively the various types of overhead that incur during dynamic scheduling. One type of overhead is the time spent accessing and operating on a shared variable; in our case loop indeces. This time is not constant in practice and it depends on several factors such as network traffic, number of simultaneous requests for a particular index and so on. For our experiments we chose this overhead to be constant and independent of the loop size or the number of processors. Since the purpose of our experiments is to study the relative (rather than the absolute) performance of GSS(1) and SS, the above assumption is not very restricting. For each scheduling decision the overhead is assumed to be a constant which represents, for instance, the number of clock cycles spent operating on a shared variable. Let o denote the overhead constant. We conducted the simulations for a best case (o_b), and a "worst" case (o_w) overhead. For the best case o_b=2 since at least two clock cycles are needed to operate on a shared variable. For the worst case we chose $o_w = 10$. In real parallel processor machines o_b and o_w can be much greater, but we are more interested in the difference $o_b - o_w$ rather than in their absolute values. E_1^b and E_1^w denote the set of experiments that ignored `if` statements for o_b=2 and $o_w = 10$ respectively. Similarly, E_2^b and E_2^w denote the set of experiments using $L1$, $L2$, $L3$, and $L4$ with `if` statements, for o_b=2 and $o_w = 10$ respectively.

The plots of Figures 7.13 and 7.14 show the speedup of the four loops $L1$ – $L4$ for different numbers of processors, for E_1 (E_1^b and E_1^w). There are four curves in each plot. Solid lines plot the speedup curves for GSS(1), and dashed lines the speedup curves for SS. More specifically the plot of Figure 7.13(a) corresponds to loop $L1$. The upper and lower solid lines are the speedup curves resulting from the schedule of $L1$ under GSS(1) and for o_b=2, o_w=10 respectively. The upper and lower dashed lines are the speedup curves of $L1$ under SS for o_b = 2, and o_w=10. The plot of Figure 7.13(b) shows the performance of GSS(1) and SS for $L2$ in E_1. Similarly Figures 7.14(a) and 7.14(b) correspond to $L3$ and $L4$ for E_1. In all plots the upper solid and dashed lines correspond to GSS(1) and SS for o_b=2 respectively. The lower solid and dashed lines correspond to GSS(1) and SS for o_w=10.

In the same way Figures 7.15 and 7.16 correspond to $L1$, $L2$, and $L3$, $L4$ respectively, for the E_2 experiments, i.e., with the `if` statements taken into account. Therefore in each plot we can see the relative performance of GSS(1), o_b=2 versus GSS(1), o_w=10; SS, o_b=2 versus SS, o_w=10; GSS(1), o_b=2 versus SS, o_b=2; and GSS(1) o_w = 10 versus SS, o_w=10, for E_1 and E_2.

Except in the case of $L3$ where both GSS(1) and SS perform almost identical, we observe that in all other cases GSS(1) is better than SS by almost a factor of two in E_1 and E_2. It is also clear from the plots that the difference in performance between GSS(1) and SS grows as the overhead grows. As it should be expected GSS(1) is less sensitive to scheduling overhead than SS.

The plots in Figures 7.17, 7.18, 7.19, and 7.20 correspond to Figures 7.13, 7.14, 7.15, and 7.16 respectively, and illustrate the speedup ratio GSS(1)/SS for each case for E_1 and E_2. The horizontal axis shows the log of the number of processors. In each plot there are two curves. The upper curve plots the speedup ratio GSS/SS for o_b=2. The lower curve plots the same ratio for o_w=10. The common characteristic of all ratio plots is that as the number of processors grows very large, the performance difference between GSS and SS becomes less significant. The large

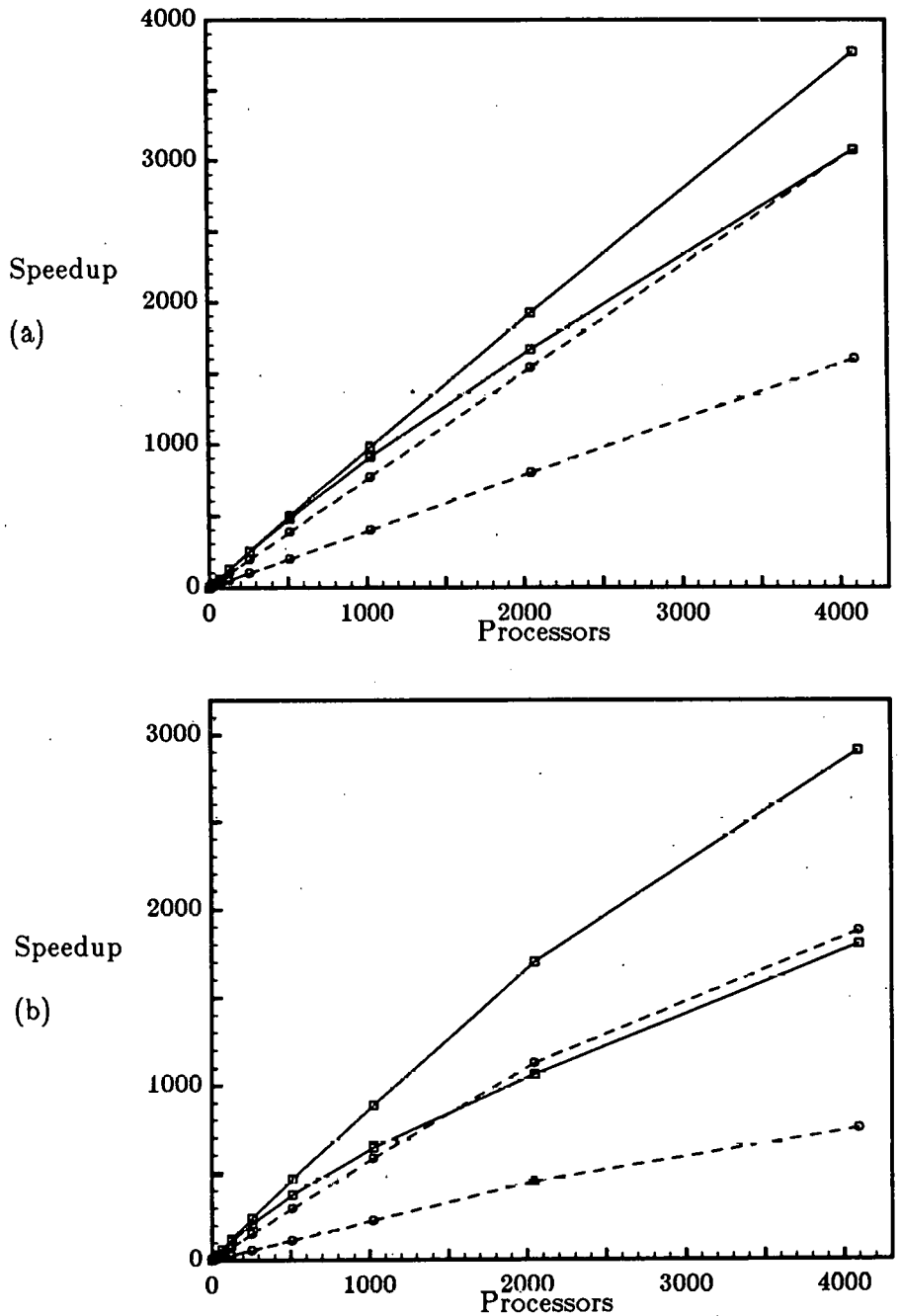


Figure 7.13. GSS and SS speedups for (a) $L1$, and (b) $L2$ without `ifs`.

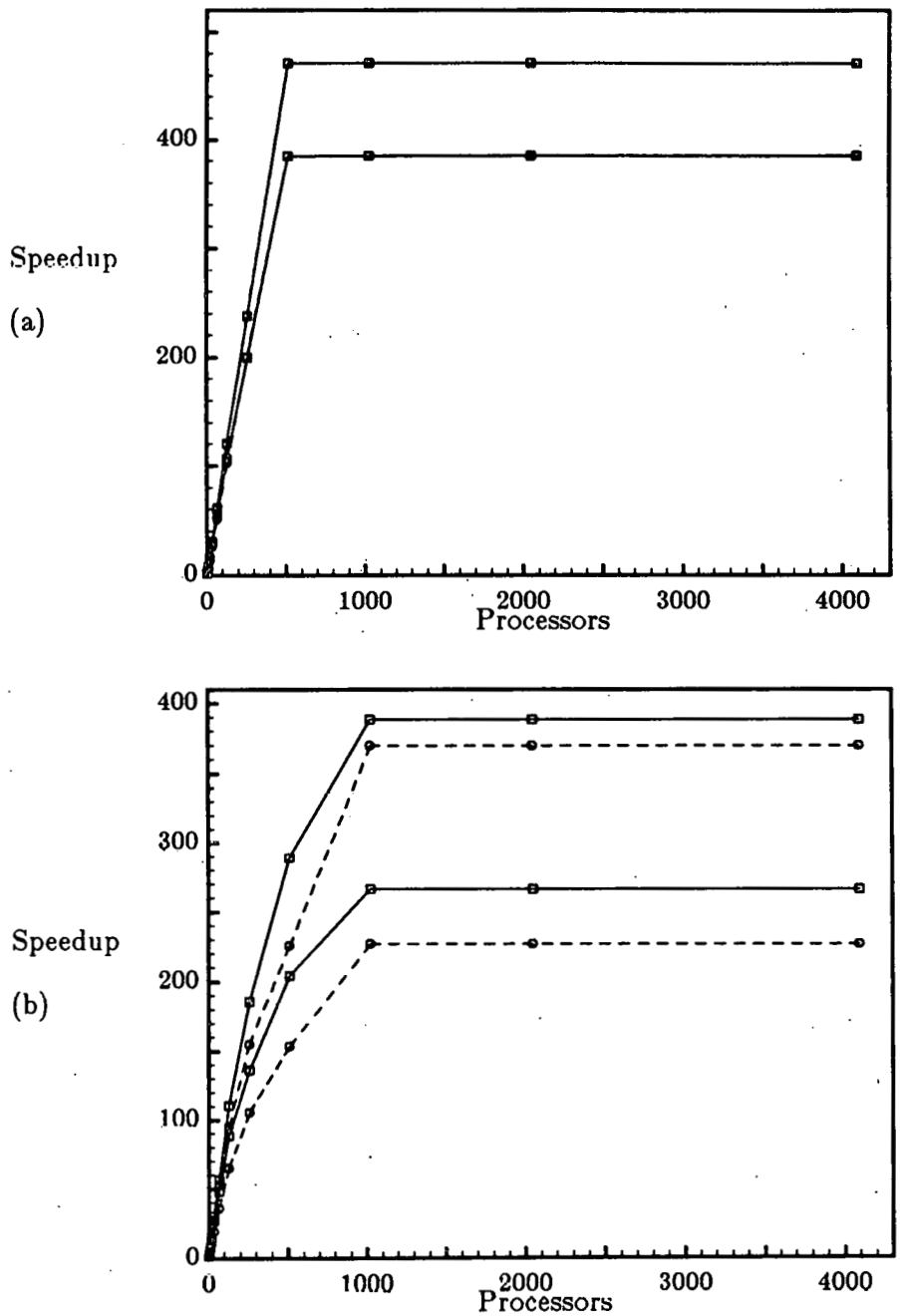


Figure 7.14. GSS and SS speedups for (a) $L3$, and (b) $L4$ without `ifs`.

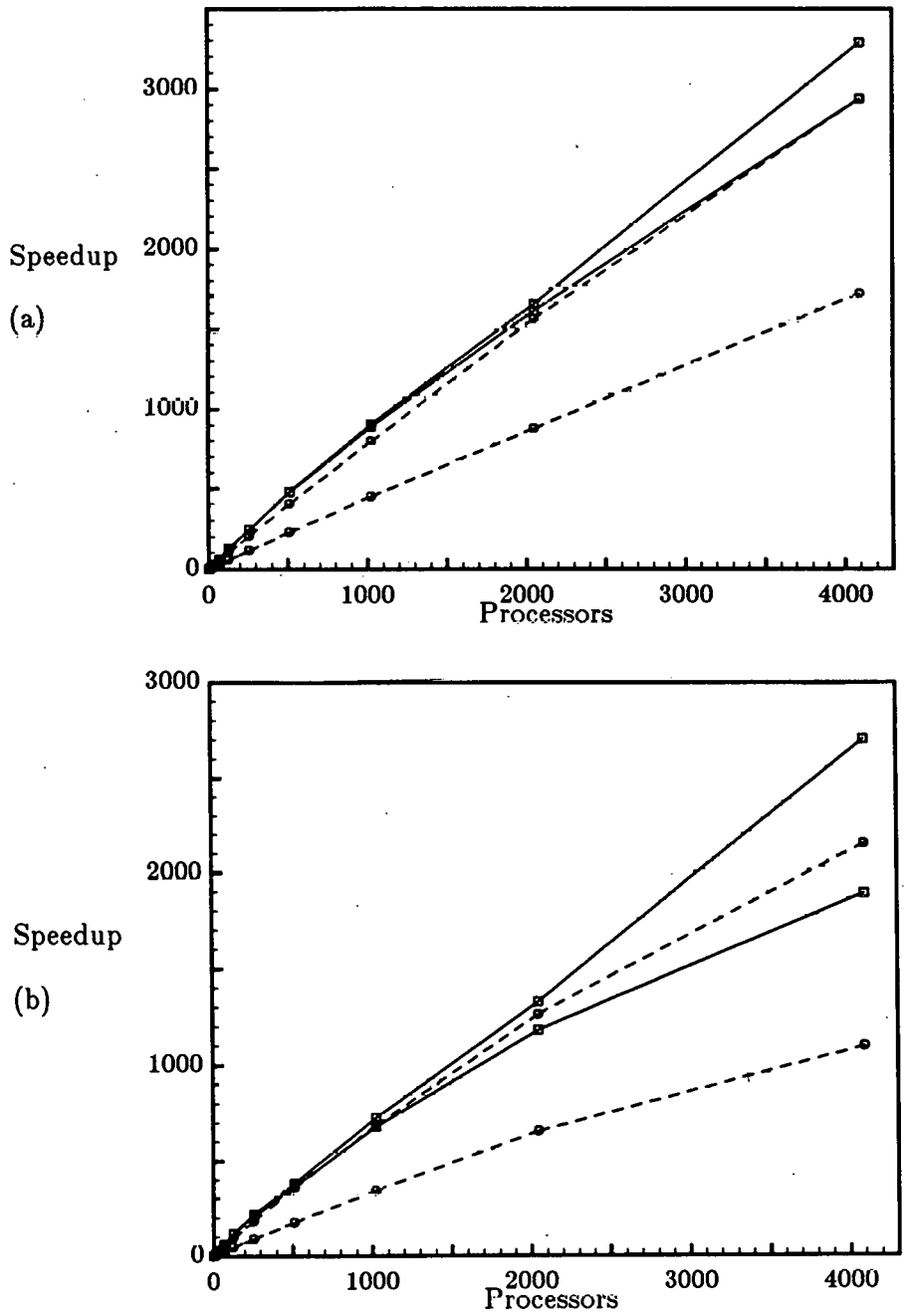


Figure 7.15. GSS and SS speedups for (a) $L1$, and (b) $L2$ with `ifs`.

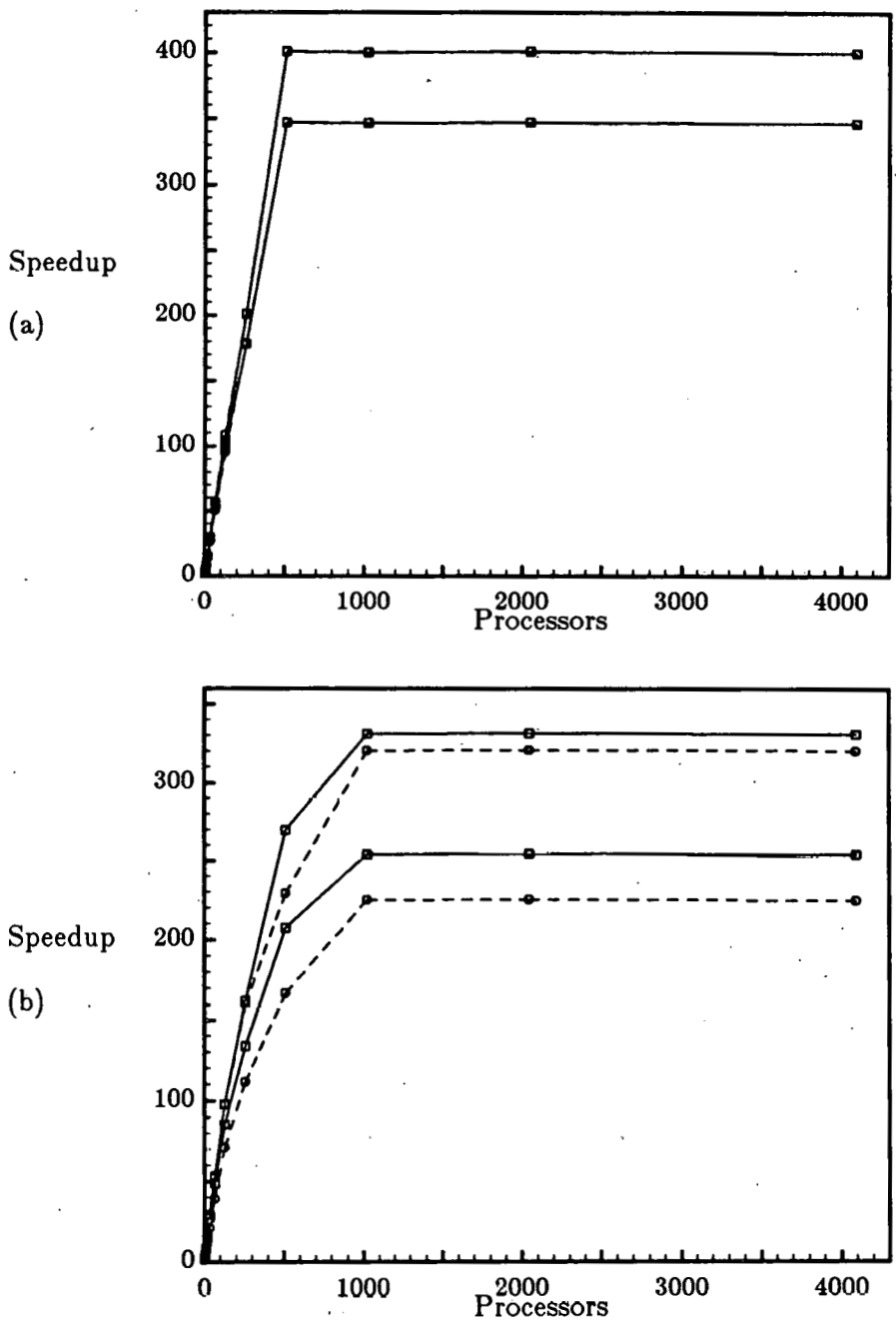


Figure 7.16. GSS and SS speedups for (a) $L3$, and (b) $L4$ with `ifs`.

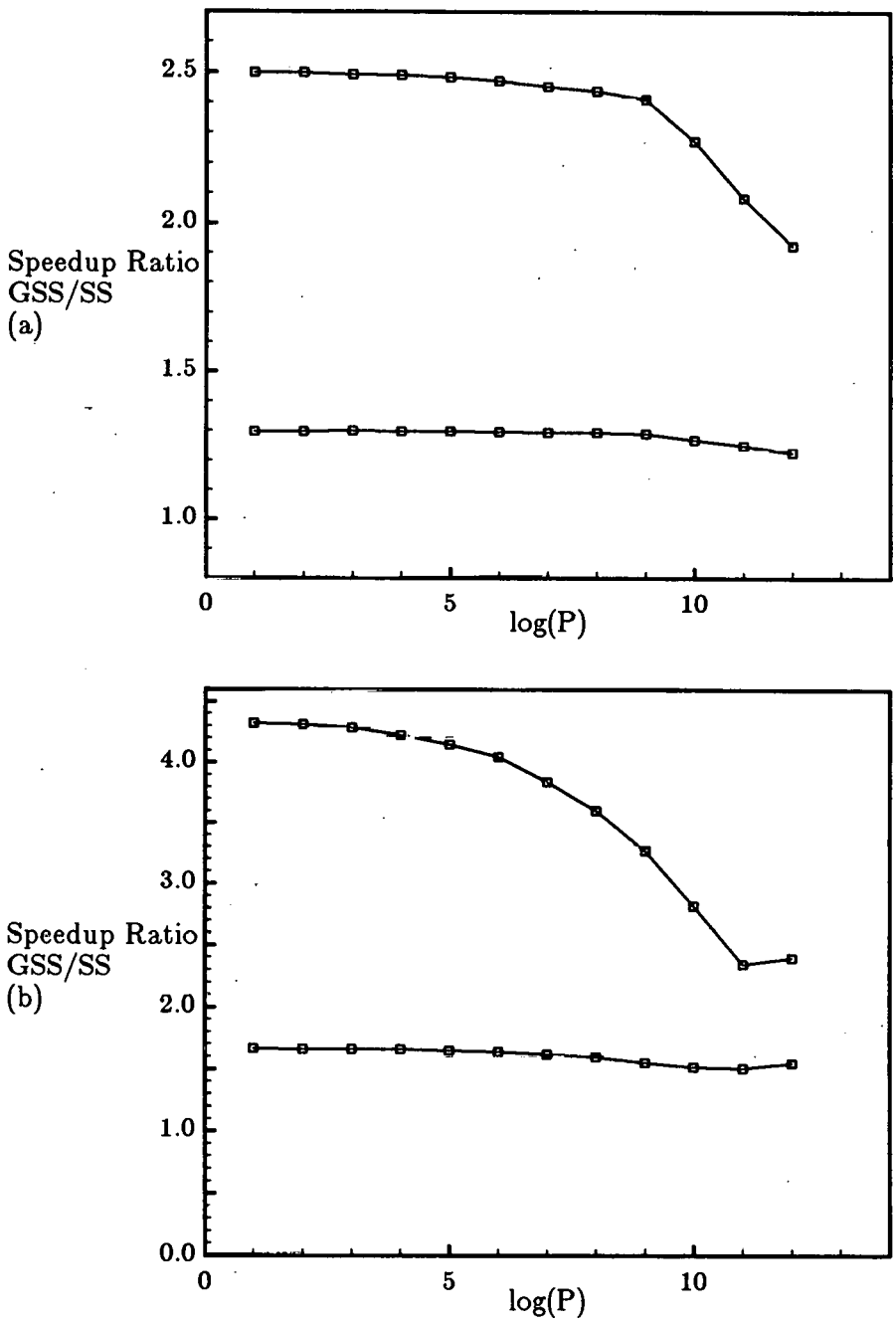


Figure 7.17. Speedup ratio of GSS/SS for (a) $L1$, and (b) $L2$ without `ifs`.

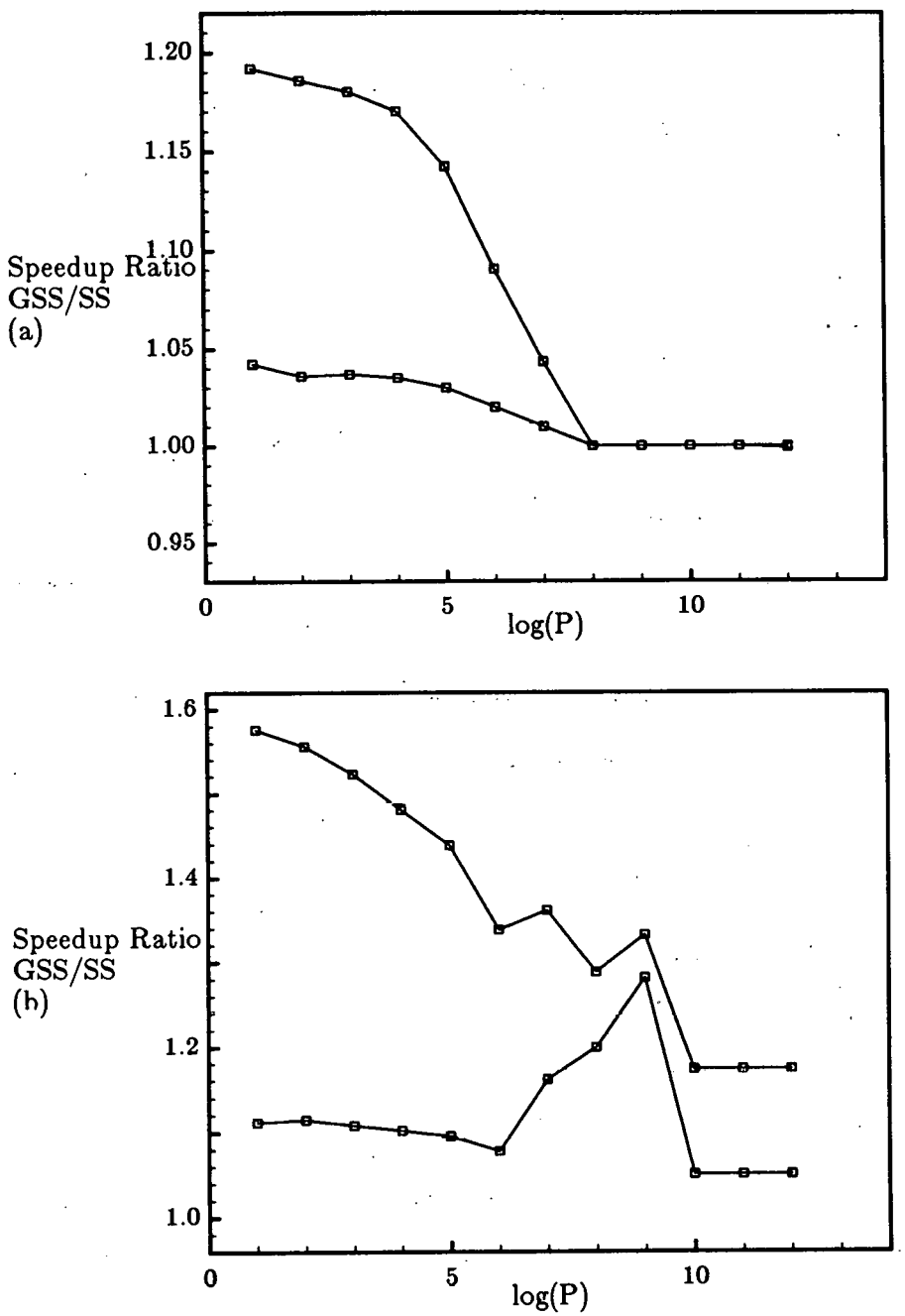


Figure 7.18. Speedup ratio of GSS/SS for (a) $L3$, and (b) $L4$ without `ifs`.

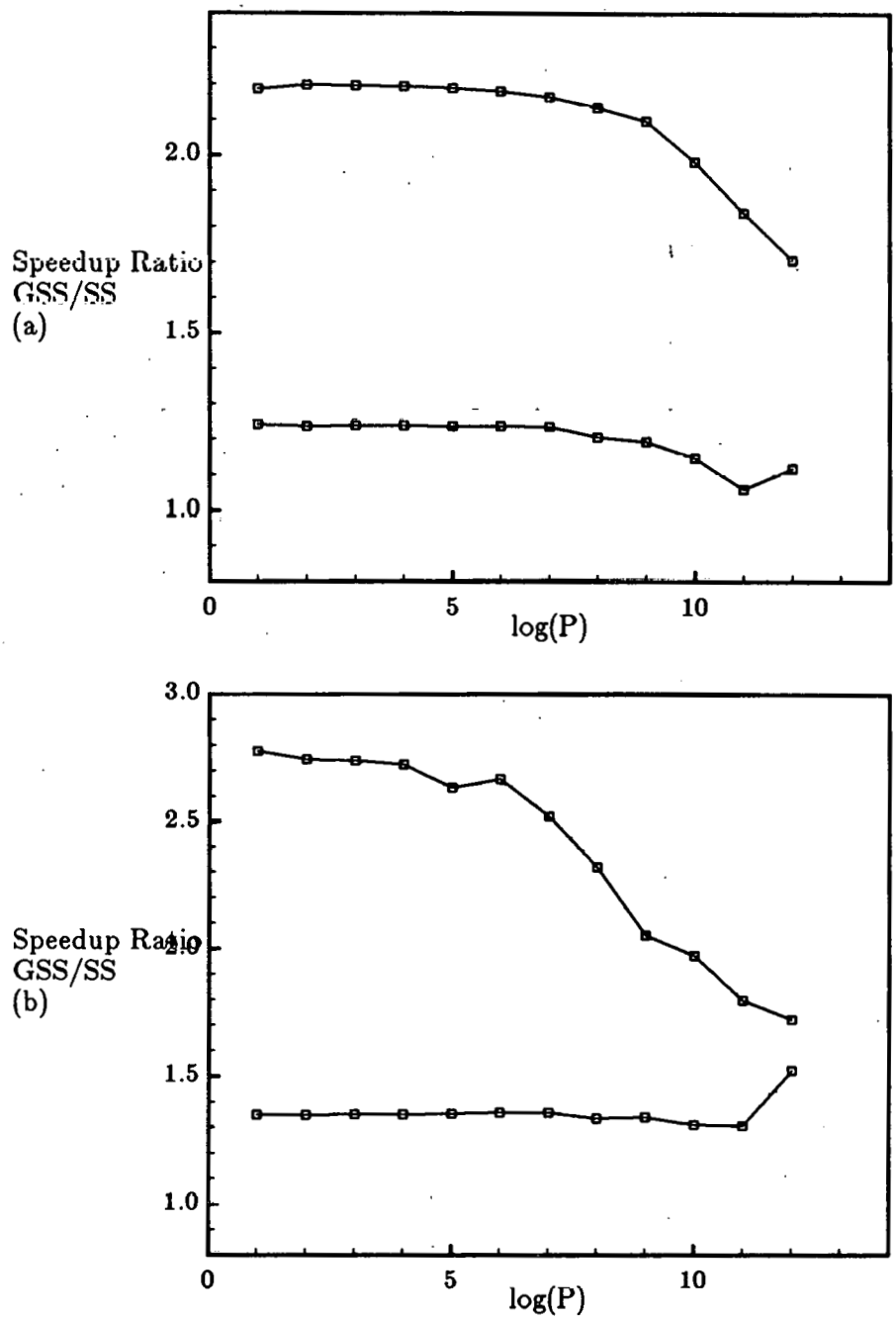


Figure 7.19. Speedup ratio of GSS/SS for (a) $L1$, and (b) $L2$ with `ifs`.

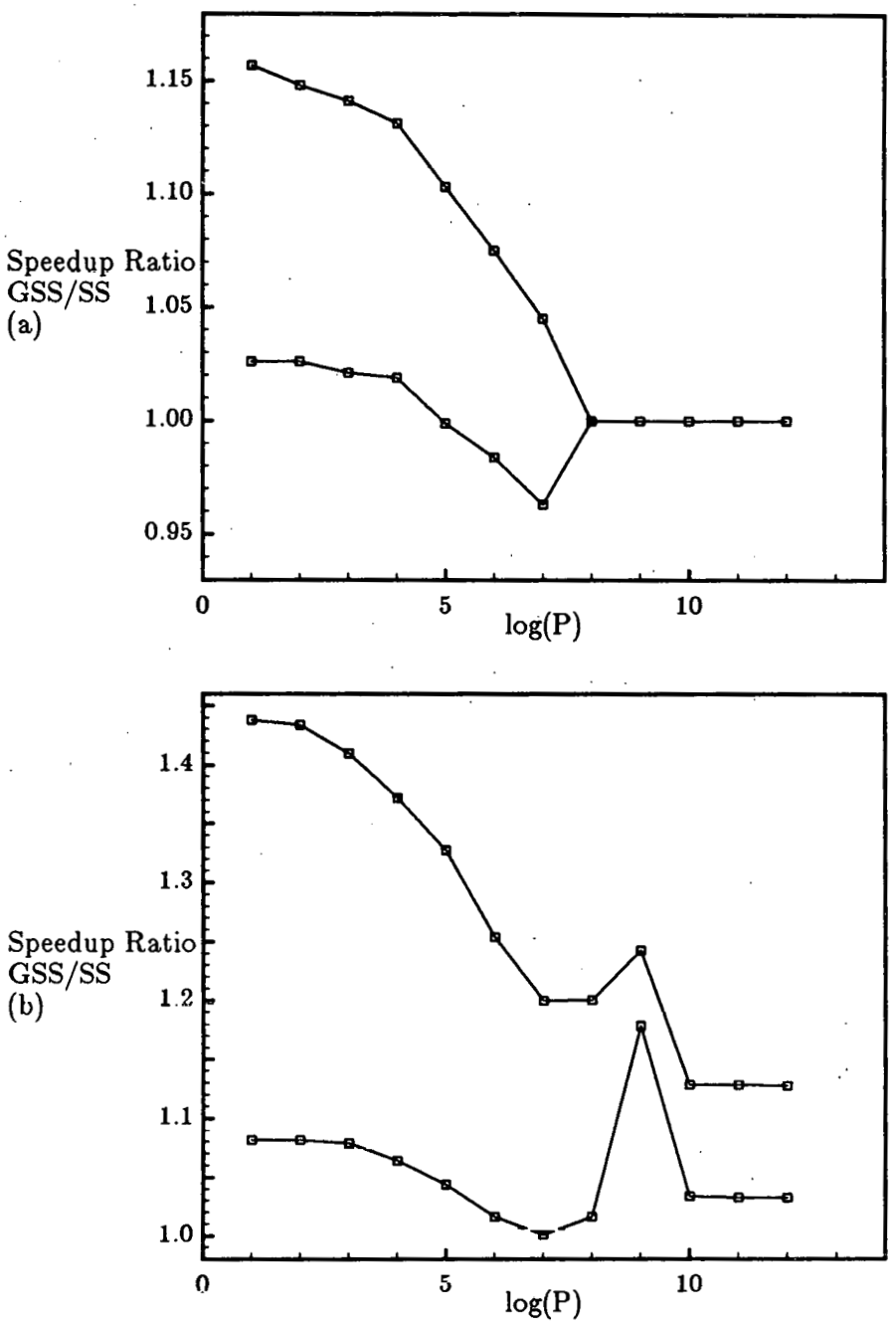


Figure 7.20. Speedup ratio of GSS/SS for (a) $L3$, and (b) $L4$ with `ifs`.

Processors	Overhead=2		Overhead=10	
	GSS	SS	GSS	SS
2	2.00	1.54	2.00	0.80
4	4.00	3.08	4.00	1.60
8	8.00	6.15	7.98	3.20
16	15.99	12.31	15.94	6.40
32	31.96	24.62	31.80	12.80
63	63.76	49.15	63.19	25.56
128	126.90	97.99	125.00	50.96
256	251.89	194.74	246.91	101.27
512	496.28	384.62	481.93	200.00
1024	975.61	769.23	909.09	400.00
2048	1923.08	1538.46	1666.67	800.00
4096	3773.58	3076.92	3076.92	1600.00

Table 7.2. GSS(1) and SS speedup values for $L1$ without `ifs`.

Processors	Overhead=2		Overhead=10	
	GSS	SS	GSS	SS
2	2.00	1.61	1.99	0.91
4	4.00	3.23	4.00	1.82
8	7.99	6.45	7.99	3.64
16	15.98	12.90	15.95	7.27
32	31.94	25.79	31.82	14.54
63	63.75	51.52	63.30	29.04
128	127.03	102.83	125.50	58.05
256	246.93	204.80	245.96	115.31
512	483.84	405.95	480.12	229.05
1024	905.00	789.02	892.07	450.05
2048	1657.38	1564.16	1614.61	878.12
4096	3289.08	2940.82	2940.82	1723.93

Table 7.3. GSS(1) and SS speedup values for $L1$ with `ifs`.

Processors	Overhead=2 GSS	Overhead=2 SS	Overhead=10 GSS	Overhead=10 SS
2	2.00	1.20	1.99	0.46
4	3.99	2.40	3.97	0.92
8	7.97	4.80	7.89	1.84
16	15.91	9.59	15.59	3.69
32	31.67	19.16	30.59	7.37
63	62.85	38.26	59.57	14.71
128	123.63	76.22	112.81	29.35
256	242.61	151.68	209.88	58.23
512	465.21	299.83	376.04	114.97
1024	884.80	582.26	644.64	228.48
2048	1702.83	1128.13	1061.76	451.25
4096	2911.29	1880.21	1805.00	752.08

Table 7.4. GSS(1) and SS speedup values for $L2$ without `ifs`.

Processors	Overhead=2 GSS	Overhead=2 SS	Overhead=10 GSS	Overhead=10 SS
2	2.00	1.48	2.00	0.72
4	4.00	2.96	3.98	1.45
8	7.99	5.91	7.92	2.89
16	15.98	11.80	15.76	5.78
32	31.94	23.57	30.42	11.54
63	63.75	46.90	61.14	22.91
128	127.03	93.42	115.78	45.88
256	246.93	184.68	211.29	90.91
512	483.84	360.23	366.02	178.04
1024	905.00	688.39	681.48	344.89
2048	1657.38	1265.04	1182.62	656.85
4096	3289.08	2157.59	1895.78	1099.68

Table 7.5. GSS(1) and SS speedup values for $L2$ with `ifs`.

Processors	Overhead=2 GSS	Overhead=2 SS	Overhead=10 GSS	Overhead=10 SS
2	2.00	1.92	1.99	1.67
4	3.99	3.85	3.95	3.33
8	7.91	7.63	7.80	6.61
16	15.54	15.02	15.20	12.99
32	30.90	30.01	29.59	25.91
63	61.12	59.95	56.18	51.55
128	120.77	110.62	106.38	102.04
256	238.10	238.10	200.00	200.00
512	471.70	471.70	384.62	384.62
1024	471.70	471.70	384.62	384.62
2048	471.70	471.70	384.62	384.62
4096	471.10	471.70	384.62	384.62

Table 7.6. GSS(1) and SS speedup values for $L3$ without `ifs`.

Processors	Overhead=2 GSS	Overhead=2 SS	Overhead=10 GSS	Overhead=10 SS
2	1.99	1.94	1.99	1.72
4	3.96	3.86	3.95	3.44
8	7.85	7.69	7.79	6.83
16	15.51	15.22	15.29	13.52
32	29.71	29.75	29.13	26.42
63	56.48	57.38	54.60	50.77
128	104.52	108.56	99.51	95.27
256	201.79	201.79	178.73	178.73
512	400.69	400.69	347.26	347.26
1024	400.08	400.08	346.74	346.74
2048	400.61	400.61	347.19	347.19
4096	399.96	399.96	346.63	346.63

Table 7.7. GSS(1) and SS speedup values for $L3$ with `ifs`.

Processors	Overhead=2 GSS	Overhead=2 SS	Overhead=10 GSS	Overhead=10 SS
2	1.99	1.79	1.97	1.25
4	3.97	3.56	3.89	2.50
8	7.89	7.12	7.60	4.99
16	15.50	14.07	14.62	9.87
32	30.34	27.68	27.53	19.13
63	56.85	52.75	48.35	36.12
128	110.83	95.34	88.99	65.32
256	186.06	155.05	136.44	105.86
512	289.62	225.74	204.67	153.50
1024	388.61	369.88	266.96	227.41
2048	388.61	369.88	266.96	227.41
4096	388.61	369.88	266.96	227.41

Table 7.8. GSS(1) and SS speedup values for $L4$ without `ifs`.

Processors	Overhead=2 GSS	Overhead=2 SS	Overhead=10 GSS	Overhead=10 SS
2	1.98	1.83	1.97	1.37
4	3.95	3.65	3.90	2.72
8	7.83	7.26	7.64	5.42
16	15.28	14.36	14.69	10.71
32	29.15	27.91	27.53	20.73
63	53.78	52.91	48.92	39.01
128	97.82	97.76	85.52	71.25
256	162.46	159.98	133.83	111.46
512	270.09	229.12	207.54	167.01
1024	331.28	320.46	254.33	225.24
2048	331.72	321.01	254.67	225.63
4096	331.12	320.54	254.22	225.29

Table 7.9. GSS(1) and SS speedup values for $L4$ with `ifs`.

perturbations in the ratio curves can be explained by the fact that GSS is "logarithmically sensi-

tive" while SS is "linearly sensitive" to scheduling overhead. Thus the performance of SS tends to saturate much earlier (as the number of processors grows) than that of GSS. As the overhead grows the improvement offered by GSS becomes more significant.

Finally Tables 7.2 - 7.9 give the detailed speedup values for the four loops of Figure 7.12 and for E_1 and E_2.

7.8. Hardware Synchronization Support for Dynamic Scheduling

The type of synchronization that is of primary importance to this work is barrier synchronization. As explained in Chapter 1 a barrier can be viewed as a program defined variable that is used as a counter. In fact many implementations of barrier synchronization use this approach [TaPe86]. Any synchronization primitive can be used to implement barrier synchronization. This approach however is costly especially when used in the context of GSS or other dynamic scheduling schemes.

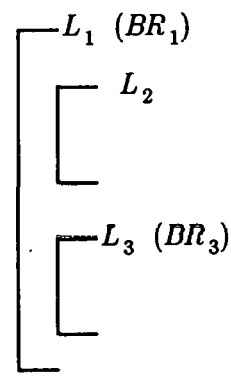


Figure 7.21. An example of hybrid loop with barriers.

In this section we propose a different implementation of barriers through the use of special bit addressable registers. Instead of full words we use single bit barriers that are *set* (1), or *cleared* (0). For example in the GSS algorithm, bit barriers can be associated with particular loops or can be inserted between loops. When a barrier associated with a loop is set, no incoming processor is allowed to dispatch new iterations. Consider for example the loop of Figure 7.21 and suppose that L_1 is serial and the other two loops are parallel. For each iteration of L_1, all iterations of L_2 and L_3 must be completed before the next iteration of L_1 can be fired. This sequencing can be achieved by associating a barrier BR_1 with L_1. The first processor to dispatch the next iteration of L_1 will set BR_1 to 1. Any incoming processors will not be allowed to proceed with L_1 until all iterations of L_3 are completed and BR_1 is cleared. We explain below how this can be done in hardware, but before we do so let us explain the disadvantages with barriers as they are implemented through the use of *P & V* primitives, the *Fetch & Add* synchronization primitives [GGKM83], or the Cedar synchronization instructions [ZhPe83].

A barrier is obviously a shared variable (counter) and thus a critical region. When several processors try to update a barrier simultaneously, all but one will be denied access to the barrier and they will be forced into busy waiting. The busy wait generates several unnecessary requests which, in the case of shared memory machines, clog up the network. This phenomenon is known as the "hot-spot" [PfNo85]. Alternatively, instead of keeping the processors busy waiting we could suspend them for a fixed amount of time and let them retry at a later moment. This can be a very inefficient solution since the entire program may wait on a single processor that has been suspended. A third more efficient solution would be to assign the processors (that are blocked on a barrier) to a new task. When that task is completed the corresponding processors will attempt to update the barrier with a much higher probability of succeeding this time. This approach would need sophisticated compile-time analysis of the program, and we plan to further

study this scheme in the future.

Here we propose another solution to the barrier synchronization problem that avoids both, hot-spots (since there is no busy waiting), and unnecessary processor latencies. The notion of barriers as shared variables is in effect eliminated. We discuss this scheme in the context of loop barriers although the solution is general. A detailed description of this approach is given in [Poly86]. In our case a barrier is not a shared variable but instead a single bit register that is writable (set) by a single processor which is determined dynamically during execution. The bit barriers are also cleared automatically by the special hardware without the intervention of the processors, or the operating system. The hardware module that implements barrier synchronization is shown in Figure 7.22. Each barrier in an *active* task is associated with such a hardware unit. Therefore at most p such units are needed in a machine with p processors. The module

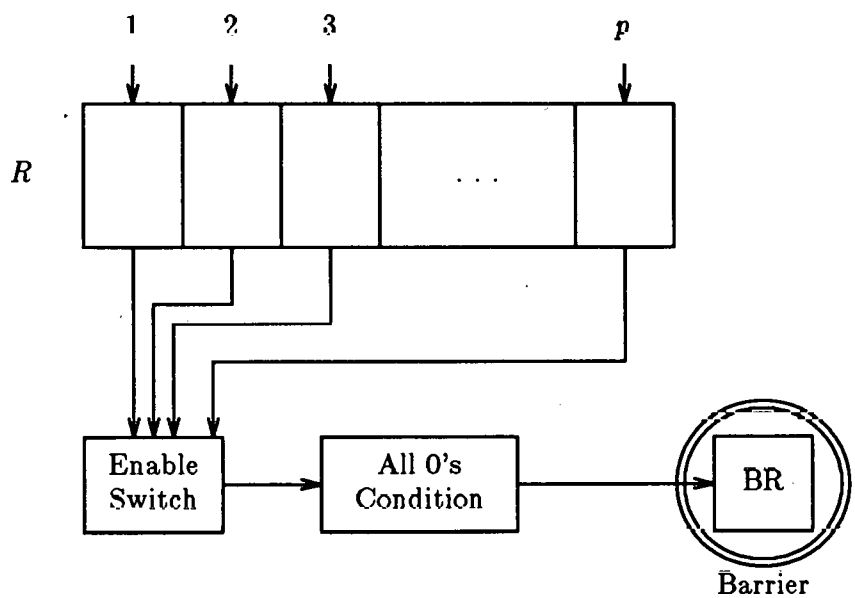


Figure 7.22. The hardware module for the implementation of barrier synchronization.

consists of a bit addressable register R, an *enable switch,* a box that checks for the *all 0's* condition in $\log_2 p$ time, and a one-bit *barrier* register BR. The R register is p bits wide and the i-th bit $R(i)$, can be set/cleared only by the i-th processor in the machine ($i = 1, 2, \ldots, p$). (This can be done in software by associating an *id* number with each processor, or by directly connecting the i-th bit of each R register to the i-th processor.) The user program can access only the BR registers and the remaining configuration is transparent to the user. BR registers are associated with loop indeces and accessing a barrier is now part of accessing a loop index (an operation that *must* be done anyway). Any conflicts that may occur now will occur during accesses to loop indeces but not to barrier variables. Therefore all the overhead associated with the explicit manipulation of software defined barriers is eliminated in this case.

Let us see how this works in the context of fork/join operations, or equivalently, in the context of parallel loops. Consider the loop of Figure 7.21 where L_1, L_2 are serial and L_3 is a parallel loop. Initially BR_1 is cleared and the first idle processor to dispatch the first iteration of L_1 sets BR_1 to 1. (Any other processors that will attempt to dispatch other iterations of L_1 will be blocked until BR_1 is cleared by the hardware.) Loop L_2 will be executed by the same processor but several processors may execute L_3. Each processor i that will dispatch part of L_3, sets the $R(i)$ bit of the corresponding register to 1. The processor that dispatches the *last* iteration(s) of L_3 will also enable the switch in the module (Figure 7.22). When a processor finishes executing on L_3, it will reset its corresponding bit in R to 0. (Note that the same bit in an R register may be set/cleared several times if the same processor dispatches several different processes of the same task.) From the time the enable switch is set on (by the last processor) the module will start testing for the all 0's condition. This can be done in a fan-in fashion and can also be pipelined so that the condition is tested every clock cycle thereafter. When the all 0's condition becomes true the BR register is cleared automatically. If the last processor to work on the task

executes for more than $\log p$ clock cycles, then this barrier synchronization mechanism involves zero overhead. The implementation of fork/join operations is straightforward. If we treat a set of independent tasks as different iterations of a parallel loop, the case of high level spreading is also identical.

Note that this hardware module can be used to implement barriers in a multiprogramming environment where tasks from different programs may be active simultaneously. If barrier modules are identified by id numbers, then each processor may be multiplexed between different processes and update the corresponding bits in the R registers in the correct order. During context switching the contents of barrier modules can be stored as part of the process state. In the case of the Cedar machine such barrier modules may be designed to be operated by the processors of a particular cluster. Separate clusters will thus have different sets of barrier modules. A processor interconnection (e.g. a bus) may allow processors to update barriers outside their cluster without going through the memory network. This can be useful when tasks are spread across many processor clusters.

This hardware implementation of barriers avoids the problems of busy waiting, synchronized accesses to critical regions, and hot-spots. In addition it eliminates the responsibility of the processors to explicitly handle barrier variables. This scheme can be used in conjunction with loop coalescing for a very efficient implementation of the GSS algorithm. The *mod* (remainder) function can be used to set loop barriers in the general case as explained in the previous section. For instance, for the loop of Figure 7.8 and the example of Section 7.7.1, the barrier associated with I_c is set each time $I_c \ mod \ (40) \ = \ 0$.

Another instance where the cost of extra hardware may be worthwhile is the case of synchronization for DOACR loops. Such loops are frequent in numerical programs [Cytr84], and conventional synchronization instructions may be very costly, especially when the loop body is

small. During execution of a DOACR loop, each iteration i passes through a "synchronization point" which is a statement that assigns, for example, a variable used by iteration $i+1$. Iteration $i+1$ may start execution only after iteration i has passed through its synchronization point. This (simple and regular) type of synchronization can be implemented in a straightforward manner through the use of a set of registers that act as a distributed bulletin board. Each processor i may have its own X_i register. All X_i registers may be written at once by any processor (broadcast operation), but each X_i may be read only by the i-th processor. All p registers contain the same value at any given moment during the execution of a DOACR. When a processor i is assigned the j-th iteration of a DOACR loop, it reads its register and the execution of that iteration proceeds if and only if $X_i = j$. When a processor executing iteration j passes through its synchronization point, it broadcasts the value $j+1$ to all X registers. Given the fact that when processor i executing iteration j must find the value j posted in X_i before it can post its own value $(j+1)$, this scheme works nicely for simple loops. The drawbacks of such an implementation however, are that it can not support the parallel execution of nested DOACRs or the concurrent execution of different DOACR loops.

7.9. Run-Time Scheduling Using a Global Control Unit

During self-scheduling the binding of processes to processors is performed dynamically. Prefetching of instructions to processors is therefore impossible. There is an exception in the case of parallel loops where processors executing part of a loop may be forced to check whether there are any iterations left before they dispatch another task. Prefetching of instructions and data to processors is meaningful only if the prefetching is performed while the processor is busy executing some other task. In order to use instruction and data prefetching consistently with each processor in the system, the scheduling should be performed by a central unit. In addition the next scheduling decision for a particular processor must be done before that processor becomes idle.

In this section we use the Weighted Priority heuristic algorithm from the previous chapter and a global control unit to perform run-time scheduling with little overhead. The scheme is simple: A global control unit is used to implement the WP heuristic at run-time. The weights may be constants or vary dynamically. The entire program graph is stored in the GCU which is then responsible for binding specific tasks to specific processors. The difference between applying the WP heuristic at run-time instead of at compile-time, is that at run-time we can obtain a more accurate estimate of the execution time and the processor demands for each task. At run-time we can also tolerate processor failures.

In the WP heuristic the program graph is transformed into a layered graph, with each layer being a set of independent tasks. A barrier synchronization (that involves all processors in the system) will have to be executed between successive layers of tasks. The tasks in each layer are grouped into subsets such that the total number of processors requested by the tasks of each subset slightly exceeds p, the total number of processors. Then processors are assigned to each subset so that each task receives a number of processors that is proportional to its size and its processor request. This scheme tends to equalize the execution time for all processors in a subset, and therefore in a layer. Since the scheduling is performed at run-time but in a rather deterministic way, scheduling overhead can be almost negligible. This is because the binding of the tasks of a given subset (or layer) to processors is decided while the processors are executing the tasks of the previous layer. In this way the instructions of the task a given processor will execute next, and its read-only data, can be prefetched before that processor completes its previous assignment. In the best case, scheduling overhead will only affect the execution of the tasks in the first layer of the program graph.

The penalty we pay in the centralized case is some idle processor time that may occur between the execution of different layers. This idle time depends solely on the structure of the

program graph and the characteristics of the code. Therefore a realistic comparison between the run-time WP scheme and the self-scheduling algorithm of the previous section can only be done on a real machine that supports both scheduling mechanisms. Since phenomena such as processor racing, network contention and overhead associated with accessing shared variables cannot be modeled accurately, simulation can be used for only approximate comparison of the two methods.

CHAPTER 8

SUBSCRIPT BLOCKING: A TRANSFORMATION FOR PARALLELIZING LOOPS WITH SUBSCRIPTED SUBSCRIPTS

There are two equally important phases in the problem of parallel processing of a given program. During the first phase the parallelism must be specified. The second phase attempts to find the best way of utilizing this parallelism on a particular machine. Several optimal and approximation algorithms that deal with this problem were presented in the previous chapters. Obviously neither step can be effective on its own. A given program can be scheduled on a parallel processor machine, given the parallel constructs in the program have been explicitly specified by the programmer, or extracted by the compiler.

There are many cases however where parallelism can not be found by the user nor by the compiler. We can distinguish these cases into feasible and non-feasible. Feasible cases usually cover problems that are very complex, and parallelism at low levels is difficult to specify of uncover. The non-feasible cases are those in which parallelism depends directly on the input and computed data, or a particular data structure makes it impossible for the compiler to extract parallelism. In such cases parallelism can only be detected and utilized at run-time. There are two possibilities for detecting and exploiting parallelism that occurs during program execution. We can either use

- the compiler, or
- synchronization instructions.

Using the compiler means having the compiler generate appropriate code that will detect and exploit parallelism when it occurs. The second alternative would be to force the task execute in parallel by synchronizing all its components. This however may involve high overhead since syn-

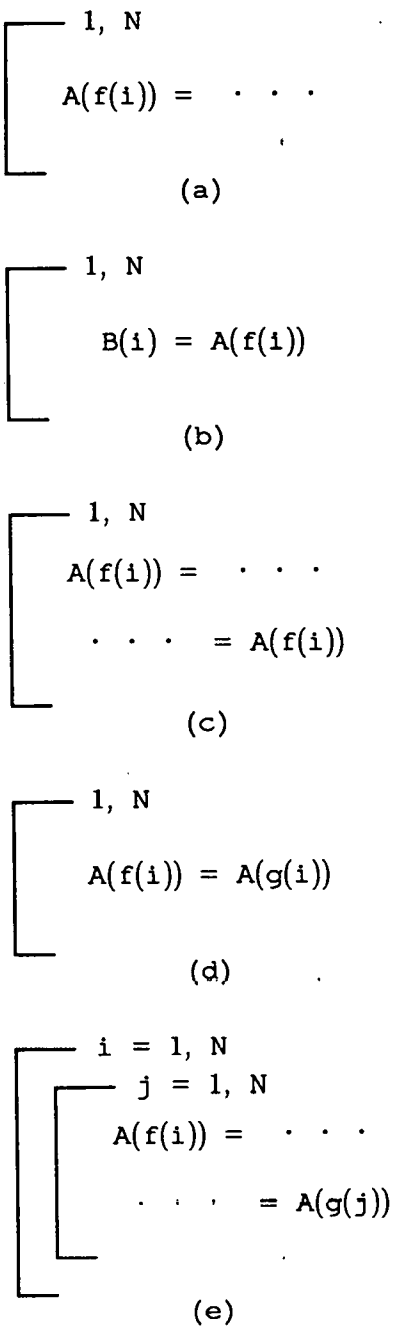


Figure 8.1. Examples of loops with subscripted subscripts

chronization is used even where it is not needed [ZhYL83].

In this chapter we consider a particular case of parallelism detection and propose a compiler transformation that solves this problem. The gain in speedup due to the resulting parallelism is also derived analytically. The next section defines the problem and discusses its different forms and use in real programs.

8.1. The Problem of Subscripted Subscripts and its Application

In Fortran programs the term *subscripted subscript* refers to a variable reference of the form $A(f(i))$ where A is the identifier of an array and its subscript $f(i)$ is itself a vector. When statements with subscripted subscripts appear in scalar code there is nothing we can do at compile-time but to assume a data dependence chain, that involves all subscripted references of the same variable. This conservative assumption would disallow potential high or low level spreading. Therefore it would result in a loss of speedup, but unless we have a large amount of scalar code the potential loss in speedup should not be significant. A similar assumption is used when subscripted subscripts appear inside loops. In such cases we assume that cross-iteration dependences of unit distance exist. This in effect serializes the corresponding loop. In such cases however the potential loss in speedup that results by serializing a loop could be very significant.

Examples of loops with subscripted subscripts are shown in Figure 8.1. In each case A denotes an array identifier and f and g subscript vectors. As mentioned in Chapter 1, two of the most vital transformations in Parafrase are the do-to-doall and do-to-doacross transformations that recognize and mark parallel loops. To do that, Parafrase performs a sophisticated dependence checking by analyzing the subscripts of array references. Dependences in scalar code are straightforward to detect. When we have array references with complicated subscripts however, (that usually occur inside loops) a detailed analysis of the subscripts must be performed to

decide whether or not a dependence of some kind exists [Bane79]. For example when two references to the same array appear inside a loop, a Diophantine equation (that involves the subscript expressions in the two references) must be solved to determine whether a dependence exists [Bane79]. Depending on the complexity of the subscript expressions, the Diophantine equation might be trivial to solve, it might be nontrivial but we could still solve it and find the exact dependences, or it might be impossible to solve. For the latter case tests exist that, although do not compute dependences, can give us an affirmative or negative answer as to whether a dependence exists. Finally there are instances (e.g. nonlinear subscript expressions) for which nothing can be done due to intractable Diophantine equations. Loops that involve such cases are of course serialized.

With loops that involve subscripted subscripts, as are the examples of Figure 8.1, the above approach cannot be applied obviously. Only in the case where the subscript $f(i)$, for example, is specified by a closed form expression a Diophantine equation can be constructed. In real cases however $f(i)$ is simply a vector of integer values that is input to the program or computed as part of the program. In such cases dependence analysis is impossible and all loops of this type are serialized. None of the existing commercial or experimental optimizing compilers parallelize general loops with subscripted subscripts.

Subscripted subscripts are heavily used in numerical programs that solve sparse systems, and in general manipulate sparse matrices, as well as in Fortran programs that implement combinatorial problems [Kuck83]. In sparse matrices only a fraction of the matrix elements are non-zero numbers. Storing the entire matrix would then be wasteful. Worse yet, if the dimension of the matrix is large the physical memory of the system might not be enough to store the entire matrix. For a $1K \times 1K$ matrix for example 8 Mbytes of physical memory would be needed if double precision is used. Alternatively, using secondary storage to store the matrix and page

it during execution would result in a significant slowdown in speed. The common approach used for storing sparse matrices is to compact the matrix and store only the non-zero elements. This is usually done by using three vectors, A, R, and C. Vector A holds the non-zero elements of a sparse matrix M, and R and C hold the row and column subscripts for each element in A. Two vectors are also used in some cases. A widely-used numerical package that solves systems of sparse equations is HARWELL. The majority of subroutines in this package contain loops with subscripted subscripts of the type shown in Figure 8.1. During operations with sparse matrices many zero elements become non-zero. This is commonly referred to as *fill-in*. Since the pattern of fill-in is unpredicatable an expandable data structure should be used to store new elements. The most popular data structures in such cases are linked lists, that expand and shrink easily. Linked lists are implemented in Fortran by means of two unbounded vectors.

In graph-theoretic problems a similar approach is used to store graphs. A graph can be represented by its adjacency matrix. The adjacency matrix is sparse if the number of edges is very small compared to the number of vertices. Large combinatorial problems are often coded in

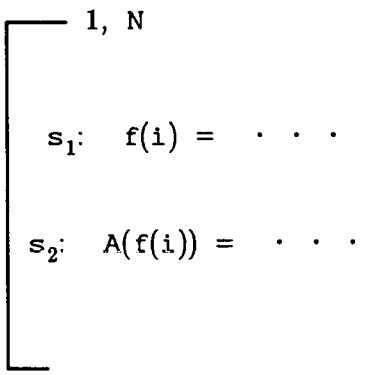


Figure 8.2. Example of definition-use of f(i).

Fortran for efficiency reasons, and their sparse matrices are store in a similar way. Parallelizing loops with subscripted subscripts would amount to a significant speedup of execution time in many cases. In this chapter we present a compiler transformation that parallelizes loops with subscripted subscripts. As shown later some overhead is introduced during the parallelization of such loops. This overhead may be nontrivial for certain loops.

The subscript blocking transformation can be implemented in the compiler internally, or as a compiler transformation. It works by examining the values of the subscript vector, ($f(i)$ in Figure 8.1 for example) and extracting the parallel iterations repeatedly. In general we distinguish two cases: 1) the subscript vector $f(i)$ is known at compile-time, and 2) $f(i)$ is computed at run-time. When $f(i)$ is known at compile-time subscript-blocking can be implicitly applied to transform the corresponding loop(s) into DOALL(s) with zero overhead. This is because, as shown later, the part of the transformation that checks the pattern of $f(i)$ is done by the compiler and it is "charged" to compilation time. For the second case where $f(i)$ is computed and it is not known at compile-time the checking of $f(i)$ must be done at run-time and that introduces some overhead. If the original loop is large enough, this overhead is amortized and is negligible compared to the benefits of the extracted parallelism. An example of subscript blocking that can be implicitly applied in the compiler is when a sparse matrix is known and $f(i)$ holds for instance, the row indeces of its non-zero elements. In this case the values of $f(i)$ are available to the compiler which can parallelize a loop in which $f(i)$ appears as a subscript, without run-time overhead. On the other hand if $f(i)$ is associated with fill-in elements which are generated during program execution, the values of the subscript vector are available only at run-time.

Before we describe subscript blocking in detail, let us consider the dependence relation between the definition of a subscript vector $f(i)$ and its use in an array reference of the form

$A(f(i))$. Consider for example the loop of Figure 8.2. If $f(i)$ is computed outside the loop that uses or assigns $A(f(i))$, subscript-blocking always works. If $f(i)$ is computed inside the loop as shown in Figure 8.2, a dependence always exists from s_1 to s_2. If there is no backward dependence from s_2 to s_1, then s_1 and s_2 belong to different π-blocks and loop distribution can thus be applied to separate the definition of $f(i)$ in s_1 and its use in s_2. If there is a hypothetical dependence from s_2 to s_1 then both statements are involved in a dependence cycle and cannot be separated. This however does not happen in practice. For instance, in all HARWELL subroutines that we examined, the definitions and uses of vector subscripts could always be distributed. Therefore, the definitions of vector subscripts are of no concern to the following material and examples.

8.2. The Transformation

As explained in Chapter 1, a dependence between two statements is represented by an arc and enforces an execution order. A statement from which a dependence originates is called a *source* and a statement to which a dependence arc points is called a *sink*. A set of successive iterations of a loop is said to form a *domain*. A *sink-domain* is a domain in which one or more statements are sinks. A domain that does not contain any sinks is called a *source-domain*. It is obvious that all source domains of a loop can be executed in parallel.

Let us consider for example the case of Figure 8.1a which involves output dependences. In this case an output dependence may exist from $A(f(i)) \rightarrow A(f(j))$ for $i > j$. If the loop involves only a single statement, we may execute it automatically as a vector statement. If we assume that memory writes are always performed in the order they are issued then the loop can be forced to execute as a vector statement without violating any dependences; only the most recent assignment for each element of A will be valid. The above assumption may be valid for SEA

```
V(1:N) = 0;
S(1) = 1;
j = 1;

DO  i=1, N
    if V(f(i))=0 then V(f(i))=1
    else  j = j+1;
          S(j) = i;
ENDO

DO  k=1,j
  DOALL  i=S(k), S(k+1)-1

         A(f(i)) = ...

  ENDOALL
ENDO
```

Figure 8.3. The subscript blocking transformation for the loop of Figure 8.1a.

systems but not necessarily for MES or MEA machines. In addition the loop of Figure 1a may be parallel but not a vector loop. In general even though the loop of Figure 1a can be vectorized for SEA machines, it should be executed serially on MES machines.

To parallelized loops of this type we apply the subscript blocking transformation which works as follows. Before we enter the loop where the vector subscript $f(i)$ is used, we examine the values of $f(i)$ and construct the "free-runs" or source domains. In other words we find subsets of successive iterations none of which contains a statement which is a dependence sink. That is, the statements of a source domain are not involved in a dependence or they are sources (originators) of output dependences. To perform the construction of source domains we use two auxiliary vectors V and S. V is a binary vector and S a vector with integer elements. Vector V is used to detect dependences (conflicts) as explained below and S holds the indeces of the

subscript-vector $f(i)$ that correspond to loop iterations that are involved in a conflict.

Specifically after $f(i)$ is generated, its elements are read and the corresponding elements of the bit-vector V are set to 1. For each $f(i)$, if $V(f(i))=0$, it is then set $V(f(i))=1$. If it is found that $V(f(i))=1$, this indicates a previous occurrence of the value $f(i)$ for another $j>i$. Since this implies a dependence from some $f(j)$ to $f(i)$, $j<i$, index i is saved in $S(k)$. All iterations up to $S(k)$-1 can therefore be executed in parallel. This procedure is performed by an extra loop that is created by the compiler before the source loop. The transformed loop of the example in Figure 8.1a is shown in Figure 8.3. Obviously the size of vectors f, V and S is equal to the size of array A. As shown in Figure 8.3 the original serial loop is transformed into a series of DOALL loops. If β is the number of times a conflict was detected, then we have a total of $\beta+1$ DOALLs created out of the original loop.

Let us consider the example of Figure 8.4. Only output dependences are considered. The values of the subscript-vector $f(i)$ are given in the top vector of Figure 8.4. After the first 3 iterations of the first loop of Figure 8.3 are executed, the 4-th, 5-th and 7-th bits of V will be set. During the next iteration for $i=4$, a conflict occurs since $V(f(4)) \neq 0$. The current index ($i=4$) is then stored in the next (2nd) empty position of S. The same process is repeated until $i=10$. The final configuration of vectors V and S is shown in Figure 8.4. The asterisks next to V indicate positions where conflicts (output dependences) occurred. The original loop is then transformed into a series of DOALLs by the second loop of Figure 8.3. In this case 4 DOALLs were created each corresponding to one of the four domains {1-3}, {4-5}, {6-9}, {10-11} of the original loop respectively.

In the above example we can observe that the creation of DOALLs was performed in a conservative manner. That is, a dependence was assumed whenever a conflict occurred, without taking into account the possibility of eliminated dependences due to the completion of earlier

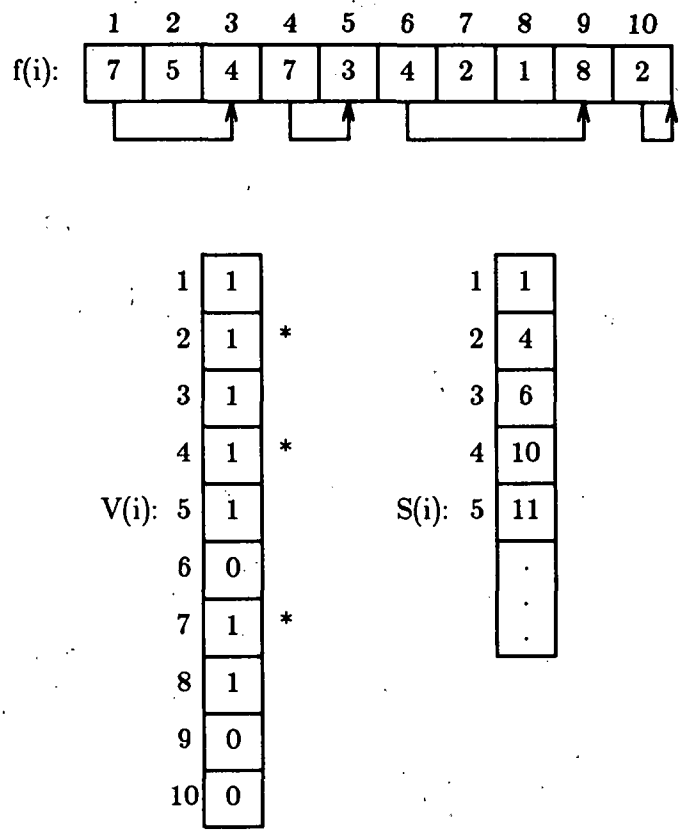


Figure 8.4. Example of computing vectors V and S.

domains. For example, in Figure 8.4, a dependence pointing to the sixth element of $f(i)$ was assumed. However the source of this dependency belongs to a previous DOALL and thus the dependence should be considered eliminated. We can solve this problem of detecting eliminated dependences by using more auxiliary storage as follows. Vector V is defined now as an integer-valued vector. Instead of setting bits in V, we store the index i in position $f(i)$ of V. Whenever a conflict occurs at $V(f(j))$ we store the corresponding index j in the next free position of S as previously and overwrite the old value of $V(f(j))$ with j. Dependences that should be ignored are specified by the test of the following lemma.

Lemma 8.1 Let m be the last index value inserted in vector S. If a conflict occurs at position $x=f(i)$ of V and $y=V(x)$, then the dependence is discarded if $y<m$. Otherwise the dependence is saved in vector S. In either case set $V(f(i)) \leftarrow i$.

Proof The proof to show that no dependence violation occurs when the above test is used is straightforward. We need to show that if a dependence is discarded by the test of the lemma, then the source of the dependence does not belong to the current domain. Let j be the position of the most recent value inserted in S, and let i be the current index of f. The current domain

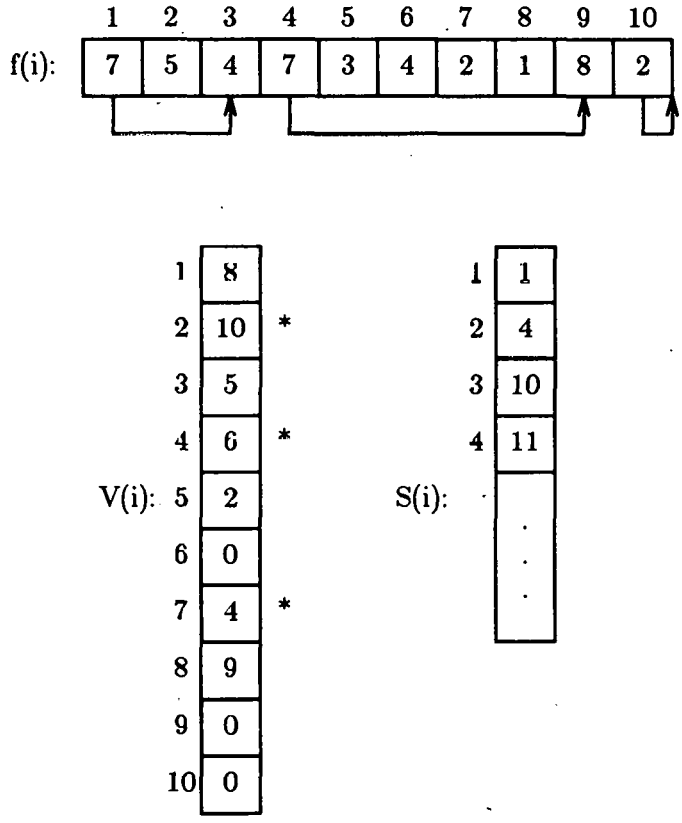


Figure 8.5. The example of Figure 8.4 when V is not a bit-vector.

includes elements from $m = f(j)$ to $f(i)$ inclusive. The conflict occurs at position $f(i)$ of V whose old value is y. Then it is obvious that the source of the dependence, i.e., y belongs to the current domain if and only if $y \geq m$, which proves the lemma. ■

The version of the example of Figure 8.4, where vector V is a vector of integers is shown in Figure 8.5. The result now is three DOALL loops. The transformation results in a series of DOALL loops as shown by the second loop of Figure 8.3. The serial loop however in this case is slightly different and is shown in Figure 8.6.

8.3. Recurrences with Subscripted Subscripts

The case of data dependences e.g., Figure 8.1c, or recurrences, e.g. Figure 8.1d, is very similar to the case of output dependences that we discussed above. Anti-dependences pose no problem if we use the following assumption. Subscript blocking transforms a serial loop into a set of DOALLs. The resulting parallel loops are executed in the order implied by the surrounding serial loop (Figure 8.3). We assume that all write operations to array A are performed in a shared copy (that resides in global memory) after the execution of a DOALL and before the next DOALL starts executing. Analogously, all elements of A referenced by a DOALL are fetched to each processor from global memory. This is a realistic assumption and if used, all anti-

```
DO  i=1, N

    if V(f(i)) >= S(j-1) then
       S(j) = i;
       j = j+1;
    V(f(i)) = i;

ENDO
```

Figure 8.6. The set-up loop for V and S when V is an integer vector.

```
V_f(1:N) = 0;
S(1) = 1;
j = 1;

DO  i=1, N
    if V_f(g(i)) > S(j) then
                j = j+1;
                S(j) = i;
    V_f(f(i)) = i;
ENDO

S(j+1) = N;

DO  k=1, j
    DOALL  i=S(k), S(k+1)-1

           A(f(i)) = A(g(i));

    ENDOALL
ENDO
```

Figure 8.7. The transformed loop of Figure 8.1d.

dependences are implicitly satisfied.

Let us consider in this section the case of subscript blocking for recurrences, and ignore for the moment output dependences. We will show how a recurrence of the type shown in Figure 8.1d can be parallelized by subscript blocking. In this case we have two subscript vectors $f(i)$ and $g(i)$. Two vectors V_f and V_g are used to carry out the tests, and a vector S to record the independent domains. As shown later the two vectors V_f and V_g are necessary only if we want to detect both data (or flow) dependences and anti-dependences. If the above assumption is used however and anti-dependences are ignored, only V_f is needed. For the sake of completeness let us use V_f and V_g and consider anti-dependences a well.

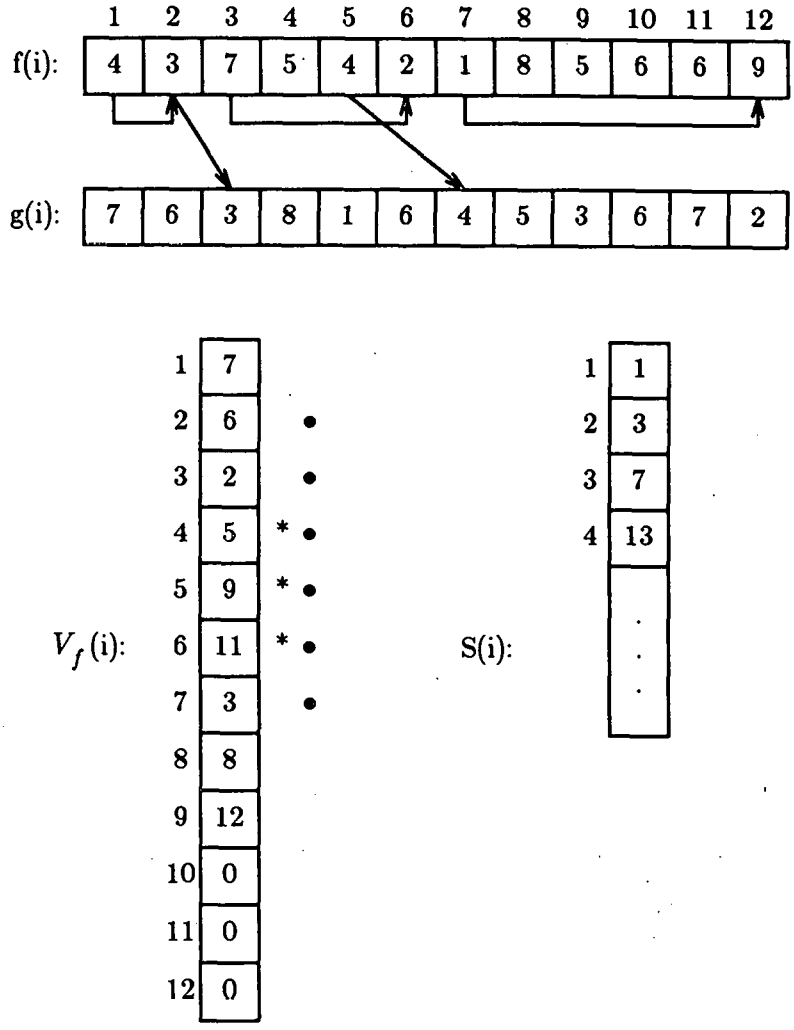


Figure 8.8. Example of subscript blocking for recurrences.

The tests in this case are similar to those of the previous section. Vector V_f is used to store the definitions of a variable and V_g to store its uses. It is clear that a flow dependence exists if a particular variable that is used is found to have a definition in the same domain, i.e., the source and the sink of a dependence belong to the same domain. More specifically, the procedure for data dependences works as follows. Initially, vector V_f is set to zero. Then starting

from $i = 1$ we examine the pairs of elements $(f(i), g(i))$ of the two subscript vectors, storing the corresponding indeces to locations $V_f(f(i))$ and $V_g(g(i))$. A data dependence is found when for some i, $V_f(g(i)) \neq 0$. The corresponding dependence is $V_f(g(i)) \rightarrow i$, and i is stored in the next free location of vector S. However as in the previous section, if the source of a flow dependence does not belong to the current domain, that dependence is correctly ignored as stated by the following lemma.

Lemma 8.2 Let j be the index of the last element of S, and $m = S(j)$. If for some i, $k = V_f(g(i)) \neq 0$ and $k < m$, the data dependence $k \rightarrow i$ is correctly discarded.

Proof The previous domains include all loop iterations up to $(m - 1)$. Since k is the source of the data dependence and $k \leq m - 1$, it belongs to a previous domain and therefore has been eliminated due to the order of execution enforced by subscript blocking. ■

As mentioned above the data dependences are detected using only vector V_f, and the corresponding domains are stored in vector S. The transformed loop of Figure 8.1d is shown in Figure 8.7. Figure 8.8 gives the vectors $f(i)$ and $g(i)$ for an application of subscript blocking using the loop of Figure 8.1d. Output dependences were ignored so far but are computed exactly as described in the previous section. The flow dependences in this example are shown in Figure

```
DO   i=1, N
     if (V_f(g(i)) or V_f(f(i)) > S(j)) then
                j = j+1;
                S(j) = i;
     V_f(f(i)) = i;
ENDO
```

Figure 8.9. The set-up loop of Figure 8.7 for data and output domains.

8.8 by arrows from $f(i)$ to $g(i)$. Dependences that were discarded by Lemma 8.2 are not shown. Asterisks and bullet-marks next to V_f indicate output and flow dependences respectively. As it can be seen, only two out of the six data dependences defined disjoint domains (DOALLs). The domains defined by data dependences alone are {1-2}, {3-6}, and {7-12}.

Anti-dependences are found in the same way using vectors V_f and V_g in the reverse order. For a given i, if $V_g(f(i)) > 0$ and $V_g(f(i))$ belongs to the current domain, an anti-dependence $A(V_g(f(i))) \rightarrow A(i)$ exists. Anti-dependences are ignored however. Output dependences can be computed here without extra storage for their corresponding domains. In fact flow and output dependences may define different domains but they can be combined to form domains that satisfy both types of dependences as shown in Figure 8.9. If an output domain lies entirely within a single flow domain, a new domain is created otherwise output domains are ignored. If in the example of Figure 8.8 both output and flow dependences are taken into consideration, the corresponding domains (DOALLs) are {1-2}, {3-6}, {7-10}, and {10-12}.

The overhead introduced by the set-up loop can be reduced by executing the set-up loop itself in parallel. For instance this can be done by using synchronization instructions to synchronize the write operations in V_f and S by each iteration. Those iterations that access

```
      DO 1 i = 1, N
         DO 2 j = 1, M
            . . .
            A(f1(i,j), f2(i,j)) = A(g1(i,j), g2(i,j))
            . . .
2        ENDO
1     ENDO
```

Figure 8.10. An example of multidimensional recurrence with subscripted subscripts.

different elements of V_f will be executed in parallel, while the writes to the same element will be done serially. Depending on how we use synchronization instructions to execute the set-up loop in parallel, we may have to order the elements of S. This can also be done in parallel. An approximate speedup bound for subscript blocking is computed in the last section of this chapter.

So far we discussed how subscript blocking can be used to parallelize singly nested loops with subscripted subscripts of any kind. In the next section we show how the same technique can be extended to parallelize multiply nested loops with subscripted subscripts.

8.4. Multiply Nested Loops

The subscript blocking transformation for loop parallelization works in precisely the same way for multiply nested loops, as it does for singly nested loops. However the auxiliary vector V now becomes a multidimensional table with a number of dimensions equal to the number of loops in the nest, plus one. Or more precisely, equal to the maximum number of subscripted subscripts in an array reference (inside the loop). Vector S can always be stored in a 2-dimensional table. If we have m nested loops for example, V will be organized as an $(m+1)$-dimensional table and S will be a 2-dimensional table with rows of size m, where each row holds values of the m indeces.

The rows of S define the boundaries of successive source domains. The result of the transformation in this case will be again two disjoint loops. The first will consist of m perfectly nested DO loops and will be functionally identical as in the single loop case (Figure 8.9). The second loop will consist of $m+1$ loops; a serial outermost loop which defines the source domains, and m DOALL loops that implement in parallel the original (untransformed) loop.

```
V_f(1:N, 1:M) = (0,0);
k=1;
S(k) = (1,1);

DO  i=1, N
    DO  j=1, M
        if (V_f(g_1(i,j),g_2(i,j)) or V_f(f_1(i,j), f_2(i,j)) > S(k)) then
                k = k+1;
                S(k) = (i,j);
        V_f(f_1(i,j), f_2(i,j)) = (i,j);
    ENDO
ENDO
S(k+1) = (N, M);
DO  l = 1, k
    n1 = S(l).1; n2 = S(l+1).1;
    m1 = S(l).2; m2 = S(l+1).2;
    if m2>1 then m2 = m2-1
    else n2 = n2-1; m2 = M;
    DOALL  i = n1, n2
        DOALL  j = m1, m2
            . . .
            A(f_1(i,j), f_2(i,j))=A(g_1(i,j), g_2(i,j));
            . . .
        ENDOALL
    ENDOALL
ENDO
```

Figure 8.11. The loop of Figure 8.10 after the transformation.

Let us see how subscript blocking works with nested loops by means of an example. For simplicity we consider the case of a 2-dimensional recurrence with subscripted subscripts as the one shown in Figure 8.10. We will apply the transformation to the loop of Figure 8.10 taking into consideration flow and output dependences only. As mentioned above, the auxiliary vector V_f is set up in this case using the same algorithm as in the previous section. Vectors V_f and S will be represented by a 3-dimensional and a 2-dimensional table, respectively. To simplify our notation and drawings we represent V_f with a 2-dimensional table where each entry can store an

ordered pair of the form (a, b), $a, b \in Z^+$. The same representation is used for S. In general, for a nested loop $L=(N_1, N_2,..., N_m)$ the size of the i-th dimension of V_f will be N_i and the size of the $m+1$-st dimension will be m. S will always be represented by a set of rows of size m.

To compare elements of V_f and S in our example we use the following rule. If (a_v, b_v) is an element of V_f, and (a_s, b_s) an element of S, then

$$(a_v, b_v) \geq (a_s, b_s)$$

if and only if $a_v \geq a_s$, or $a_v = a_s$ and $b_v \geq b_s$. If (a_s, b_s) is the i-th element (row) of S then $a_s = S(i).1$, and $b_s = S(i).2$. The transformed loop of Figure 8.10 is shown in Figure 8.11.

A detailed example for $N=3$, and $M=4$ is shown in Figure 8.12. Figure 8.12 shows the unrolled version of the example loop with arcs illustrating flow and output dependences. After the set-up loop of Figure 8.11 is executed the final configuration of table V_f and the resulting domains in S are also shown in Figure 8.12. The four domains that were created by subscript blocking in this case are {(1,1), (1,2), (1,3)}, {(1,4), (2,1)}, {(2,2), (2,3), (2,4)}, and {(3,1), (3,2), (3,3), (3,4)}. Note that from a total of six flow and output dependences only three were used to define the source domains, and the remaining were ignored.

8.5. Expected Speedup

The set-up loops created by subscript blocking can be executed in parallel using, for instance, the Cedar synchronization scheme [ZhPe83]. In case the subscript vector is known at compile-time the transformation can be applied without the set-up loop, in which case the extra overhead is zero. We believe that this rarely happens in real code and therefore the set-up loop is needed to define the domains at run-time. The body of the set-up loop will always contain three to four statements, and in general it is independent of the body size of the original loop.

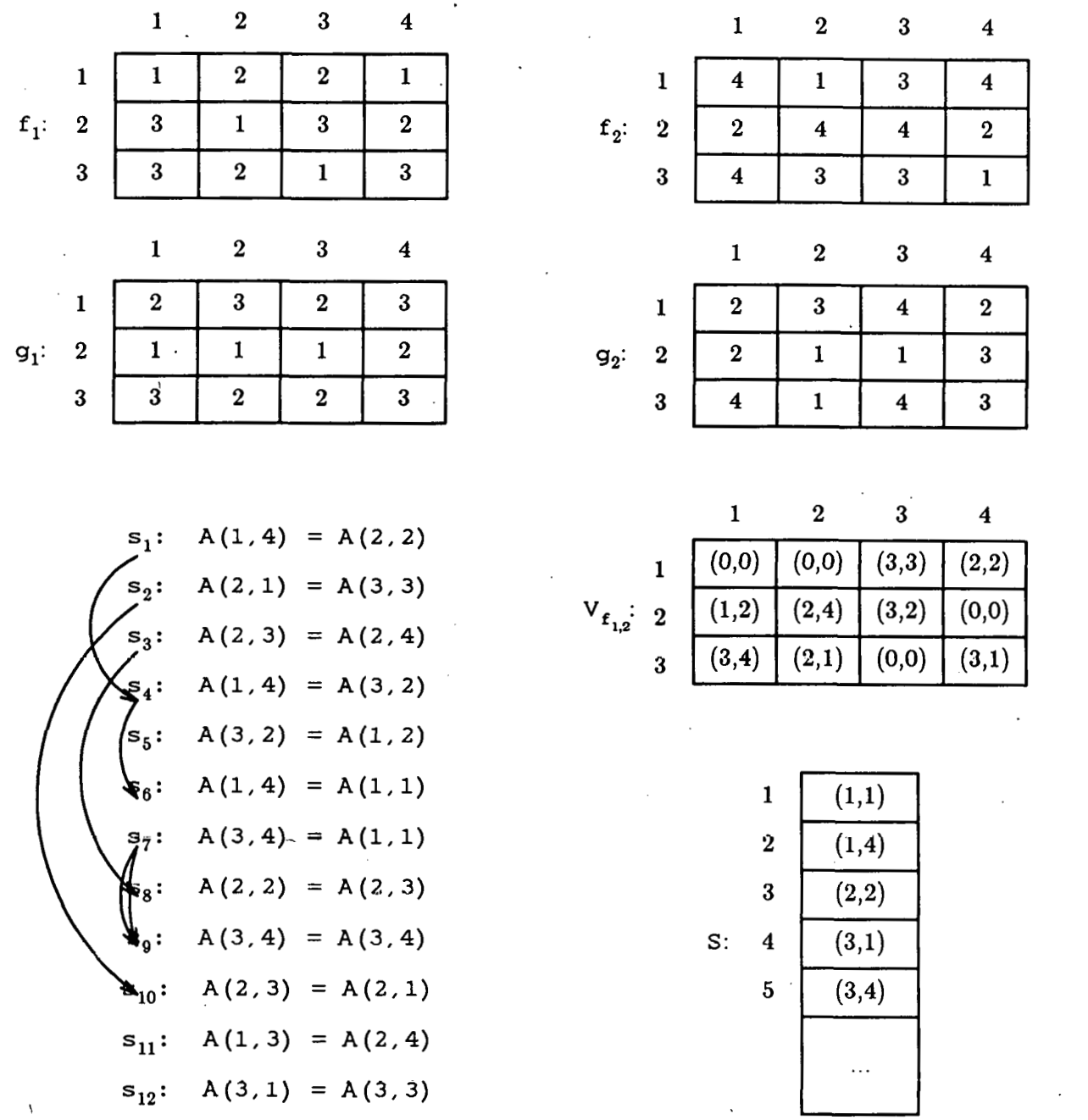


Figure 8.12. The unrolled loop of Figure 8.10 with its subscript values and the V_f and S tables.

Let C and B be the execution time of the loop body (i.e., one iteration) of the set-up loop, and the original loop respectively. If we assume that each statement takes a unit of time to execute, then we always have $C=3$, or 4. Consider the case of a perfectly nested serial loop $L=(N_1, N_2,..., N_m)(B)$ of nest depth m, where N_i is the number of iterations of the i-th loop and B is the execution time of the loop body. Subscript blocking will transform this loop into two loops $O=(N_1, N_2,..., N_m)(C)$ and $R=(\beta, N_1, N_2,..., N_m)(B)$. O is the set-up loop and R is a set of m DOALL loops nested in a serial outermost loop with β iterations. β is the number of domains. Let T_p^O, T_p^R be the parallel execution times of O and R on p processors, respectively. Also let $N = \prod_{i=1}^{m} N_i$. Since the execution time of the original loop is NB, the expected speedup of the transformed loop on p processors would be

$$S_p = \frac{NB}{T_p^O + T_p^R}. \tag{8.1}$$

Let α_i be the number of writes (conflicts) to the i-th element of V_f, and $\alpha=\max_i \{\alpha_i\}$. Since the updates to the same element of V_f are serialized, the parallel execution time of O with unlimited processors will be determined by the maximum number of conflicts in each element of V_f, i.e., $T_\infty^O = \alpha C$. On p processors, each processor will execute an average of $\lceil N/p \rceil$ iterations of O. Therefore for the limited processor case,

$$T_p^O = \begin{cases} \left\lceil \frac{N}{p} \right\rceil C & \text{if } \alpha \leq \left\lceil \frac{N}{p} \right\rceil \\ \alpha C & \text{otherwise} \end{cases} \tag{8.2}$$

T_p^R is computed as follows. Let β be the number of domains and m_i be the number of iterations in the i-th domain, ($i=1, 2,..., \beta$). Obviously $\sum_{i=1}^{\beta} m_i = N$. Then we have

$$T_p^R = \sum_{i=1}^{\beta} \left\lceil \frac{m_i}{p} \right\rceil B \tag{8.3}$$

and since by definition

$$\frac{m_i}{p} \leq \left\lceil \frac{m_i}{p} \right\rceil \leq \frac{m_i}{p} + 1, \quad (i = 1, 2, \ldots, \beta) \tag{8.4}$$

from (8.3) and (8.4) it follows that

$$\frac{NB}{p} \leq T_p^R \leq \frac{NB}{p} + \beta B \tag{8.5}$$

If we assume $T_p^O \approx NC/p$ in (8.2), we finally have from (8.1) and (8.5) that

$$\frac{NB}{(NB)/p + (NC)/p} \geq S_p \geq \frac{NB}{(NB)/p + (NC)/p + \beta B} \quad \text{or,}$$

$$p\left(\frac{B}{B + C}\right) \geq S_p \geq p\left(\frac{B}{B + C + (\beta B p)/N}\right). \tag{8.6}$$

When N is large relative to β and p, the speedup converges to the upper limit in (8.6). As an example, for a loop with $N=100$, $B=50$, $C=3$, $p=16$ and $\beta=8$, the speedup range (depending on the size of the domains) is

$$15 \geq S_{16} \geq 7.$$

If all eight domains have 16 or fewer iterations, then $S_{16} \approx 12.5$. We plan to implement subscript blocking in Parafrase and measure its effect on sparse matrix solvers, where subscripted subscripts appear frequently.

CHAPTER 9

CONCLUSIONS

The speed of computer systems grew by approximately a factor of ten every three years until the end of the last decade, and this was largely due to improvements in device technology. Although increased performance is still achieved through improvements in technology and the advent of new technologies, the factor of speed-increase as a function of time is only a fraction of what it used to be. In recent years, attention has focused on alternative sources for increasing computer performance. The major source is parallelism in its many (ambiguously defined) forms. Although the basic principles of parallelism are old, a systematic research effort in the area of parallel processing has only begun. Of course this was made possible through advancements in technology that allow us to design fast, compact, and cost-effective components.

As opposed to technology (which offers increased performance through improvements in hardware), parallelism can be applied to algorithms, languages, and hardware. It is a more general, and potentially more powerful alternative for increasing performance without theoretical limits. There are many crucial problems involved in specifying, extracting, and exploiting parallelism. Most of these are very complex problems, but their solution is a "must" for the realization of large scale parallel processor systems.

In this thesis we investigated and proposed solutions to many important problems in parallel processing. We define the notion of parallelism in general for program models and in particular for Fortran programs. We discuss the different types of parallelism and develop a notation that is useful for describing the mode of execution for a set of serial and parallel tasks. We considered the problem of interprocessor communication and proposed an optimal solution for special types of program graphs.

The problem of processor allocation and scheduling for arbitrarily nested parallel loops was discussed in great detail. Modern parallel processor systems can exploit up to two dimensions of parallelism in Fortran programs. In this thesis we developed analytical methods for studying multidimensional parallelism (multiply-nested loops) and proposed optimal static algorithms for the general case. An optimal self-scheduling algorithm was also presented. This scheme involves less overhead than any other known self-scheduling scheme. Analytical and simulation results showed the advantage of GSS over self-scheduling. Two models for estimating run-time overhead were developed for the case of linear and logarithmic (on the number of processors) overhead. These models can be used to predict the optimal number of processors for a given task before the actual assignment of processors is performed.

Heuristic and optimal algorithms were developed for high-level spreading and processor allocation to general program graphs. Finally we presented two new compiler transformations: loop coalescing, and subscript blocking. The former transformation is useful for static and dynamic scheduling. The latter transformation can be used to vectorize previously unvectorizable loops with subscripted subscripts. When the original loop limits are large enough, the resulting speedup can be proportional to the number of processors.

This dissertation proposes new solutions to some important problems in parallel processing. The next step in our future work will involve the extension of the algorithms and techniques presented in this thesis to include memory allocation and related compiler optimizations, and their implementation. Specialized hardware for the fast implementation of the proposed self-scheduling scheme along with a sophisticated restructuring compiler, would be a powerful combination for solving the major problems in parallel processing: scheduling, communication, and synchronization.

REFERENCES

[AhUl77] A.V. Aho and J.D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Massachusetts, 1977.

[Alli85] Alliant Computer Systems Corp., "FX/Series Architecture Manual," Acton, Massachusetts, 1985

[Amda67] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Computer Conference Proc.,* Vol. 30, 1967.

[Bain78] W.L. Bain, "Hardware Scheduling Strategies for Systems with Many Processors," *Inter. Conf. on Parallel Processing,* August, 1979.

[Bane76] U. Banerjee, "Data Dependence in Ordinary Programs," M.S. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-76-837, November, 1976.

[Bane79] U. Banerjee, "Speedup of Ordinary Programs," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-79-989, October 1979.

[Bane81] U. Banerjee, "Bounds on the Parallel Processor Speedup," *Proc. of 19th Annual Allerton Conference on Communication, Control, and Computing,* September-October, 1981.

[Bokh85] S. Bokhari, "Partitioning Problems in Parallel, Pipelined and Distributed Computing," NASA ICASE Report No. 85-54, Langley Research Center, 1985.

[BuSi85] I. Bucher, M. Simmons, "Performance Assessment of Supercomputers," *Los Alamos National Lab., LA-UR-85-1505,* 1985.

[Chen83] S. Chen, "Large-scale and High-speed Multiprocessor System for Scientific Applications - Cray-X-MP-2 Series," *Proc. of NATO Advanced Research Workshop on High Speed Computing, Kawalik(Editor),* pp. 59-67, June 1983.

[CGJ 78] E.G. Coffman, M.R. Garey, D.S. Johnson, "An Application of Bin-Packing to Multiprocessor Scheduling," *SIAM J. Comput.,* Vol. 7, No. 1, February, 1978.

[Coff76] E.G. Coffman, Jr., ed., *Computer and Job-shop Scheduling Theory,* John Wiley and Sons, New York, 1976.

[CoGr72] E.G. Coffman and R.L. Graham, "Optimal Scheduling on Two Processor Systems," *Acta Informatica,* Vol. 1, No. 3, 1972.

[Cray85] "Multitasking User Guide," Cray Computer Systems Technical Note, SN-0222, January, 1985.

[Cytr84] R.G. Cytron, "Compile-time Scheduling and Optimization for Asynchronous Machines," Ph.D. Thesis, University of Illinois at Urbana Champaign, DCS Report No. UIUCDCS-R-84-1177, 1984.

[Cytr86] R.G. Cytron, "Doacross: Beyond Vectorization for Multiprocessors (Entended Abstract)," *Proceedings of the 1986 International Conference on Parallel Processing,* St. Charles, IL, pp. 836-844, August, 1986.

[DavJ81] J. R. Beckman Davies, "Parallel Loop Constructs for Multiprocessors," M.S. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-81-1070, May, 1981.

[DBMS79] J. J. Dongarra, J.R. Bunch, C. B. Moler, and G. W. Stewart, "Linpack User's Guide," *SIAM Press,* Philadelphia, PA, 1979.

[Dong85] J. J. Dongarra, "Comparison of the CRAY X-MP-4, Fujitsu VP-200, and Hitachi S-810/20: An Argonne Perspective," Argonne National Laboratory, ANL-85-19, October, 1985.

[FlHe80] M. J. Flynn, J. L. Hennessy, "Parallelism and Representation Problems in Distributed Systems," *IEEE Trans. on Computers,* C-9, 1980.

[GaJo79] M.R. Garey and D.S. Johnson, "Computers and Intractability, A Guide to the Theory of NP-Completeness," W.H. Freeman and Company, San Francisco, California, 1979.

[GGKM83] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer -- Designing an MIMD Shared-Memory Parallel Machine," *IEEE Trans. on Computers,* Vol. C-32, No. 2, pp. 175-189, February 1983.

[GKLS83] D. Gajski, D. Kuck, D. Lawrie and A. Sameh, "CEDAR -- A Large Scale Multiprocessor," *Proc. of International Conference on Parallel Processing,* pp. 524-529, August 1983.

[Gonz77] M. J. Gonzalez, "Deterministic Processor Scheduling," *Computing Surveys,* Vol. 9, No. 3, Sept., 1977.

[Grah72] R. L. Graham, "Bounds on Multiprocessor Sceduling Anomalies and Related Packing Algorithms," *Spring Joint Computer Conf.,* 1972.

[Jack85] D. T. Jackson, "Data Movement in Doall Loops," M.S. Thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Report No. CSRD-524, 1985.

[Hu 61] T. C. Hu, "Parallel Sequencing and Assembly Line Problem," *Oper. Research,* Vol. 9, Nov., 1961.

[Husm86] H. Husmann, Ph.D. Thesis, Center for Supercomputing Research and Development, University of Illinois at Champaign-Urbana, 1986.

[IEEE79] "Programs for Digital Signal Processing," Edited by Digital Signal Processing Committee, IEEE Press, N. Y., 1979.

[KaNa84] H. Kasahara and N. Seinosuke, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. Compt.,* Vol. C-33, No. 11, Nov., 1984.

[Kenn80] K. Kennedy, "Automatic Vectorization of Fortran Programs to Vector Form," Technical Report, Rice University, Houston, TX, October, 1980.

[KKLW80] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Fourth International Computer Software and Applications Conference,* October, 1980.

[KLPL81] D.J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Proceedings of the 8-th ACM Symposium on Principles of Programming Languages,* pp. 207-218, January 1981.

[KLSG83] D. J. Kuck, D. H. Lawrie, A. Sameh, and D. Gajski, "The Architecture and Programming of the Cedar System," *Proceedings of the 1983 LASL Workshop on Vector and Parallel Processing,* Los Alamos, New Mexico, August, 1983.

[Knut81] D. Knuth, *The Art of Computer Programming,* Volumes 1 and 2, Second Edition, Addison Wesley, 1981.

[Koba81] H. Kobayashi, *Modeling and Analysis,* Second Edition, Addison Wesley, 1981.

[Krus84] C. Kruskal, "An Optimistic View on Speedup Bounds," Lecture Notes, NATO Summer School, West Germany, July, 1984.

[KrWe85] C. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Trans. on Software Engineering,* Vol. SE-11, No. 10, October, 1985.

[Kuck73] D.J. Kuck et al, "Measurements of Parallelism in Ordinary FORTRAN Programs," *Proc. 1972 Sagamore Computer Conf. on Parallel Processing,* 1972.

[Kuck78] D.J. Kuck, *The Structure of Computers and Computations,* Volume 1, John Wiley and Sons, New York, 1978.

[Kuck84] D.J. Kuck et. al., "The Effects of Program Restructuring, Algorithm Change and Architecture Choice on Program Performance," *International Conference on Parallel Processing,* August, 1984.

[Kuck83] D. J. Kuck, Unpublished Memo on Sparse Matrices and Manipulation of Subscripted Subscripts, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1983.

[Lee 80] R. B. Lee, "Empirical Results on the Speedup, Efficiency, Redundancy, and Quality of Parallel Computations," *Inter. Conf. on Parallel Processing,* 1980.

[LeKK86] G. Lee, C. Kruskal, and D. J. Kuck, "The Effectiveness of Combining in Shared Memory Parallel Computers in the Presence of 'Hot Spot'," *Proceedings of the 1986 International Conference on Parallel Processing,* St. Charles, IL, August, 1986.

[Liu 81] C.L. Liu, "Combinatorial Problems and Approximation Solution," *Current Trends in Programming Methodology,* M. Chandy, Edit., Prentice-Hall Inc., 1981.

[LuBa80] S.F. Lundstron and G.H. Barnes, "Controllable MIMD Architecture," *Proceedings of the 1980 Conference on Parallel Processing,* pp. 19-27, 1980.

[Mann84] R. Manner, "Hardware Task/Processor Scheduling in a Polyprocessor Environment," *IEEE Trans. Compt.,* Vol. C-33, No. 7, July, 1984.

[MiPa86] S. P. Midkiff and D. A. Padua, "Compiler Generated Synchronization for DO loops," *Proc. of the 1986 International Conference on Parallel Processing,* St. Charles, IL, pp 544-551, August, 1986.

[Mins70] M. Minsky, "Form and Computer Science," *JACM,* Vol. 17, 1970.

[MuCo69] R. R. Muntz and E. G. Coffman, "Optimal Preemptive Scheduling on Two-Processor System," *IEEE Trans. Compt.,* Vol. C-18, No. 11, Nov., 1969.

[Nett76] E. Nett, "On Further Applications of the Hu Algorithm to Scheduling Problems," *Inter. Conf. on Parallel Processing,* 1976.

[Padu79] D.A. Padua Haiek, "Multiprocessors: Discussions of Some Theoretical and Practical Problems," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCDCS-R-79-990, November 1979.

[PaKL80] D. A. Padua, D. J. Kuck, D. H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. on Computers,* Vol C-24, No. 9, pp.. 763-776, Sept., 1980.

[PfNo85] G. F. Pfister, V. A. Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks," *Proceedings of the 1985 International Conference on Parallel Processing,* St. Charles, IL, August, 1985.

[PoBa86] C. D. Polychronopoulos, U. Banerjee, "Speedup Bounds and Processor Allocation of Parallel Programs on Multiprocessor Systems," *Proceedings of the 1986 International Conference on Parallel Processing,* St. Charles, IL, pp. 961-968, August, 1986.

[PoKP 86] C. D. Polychronopoulos, D. J. Kuck, D. A. Padua, "Optimal Processor Allocation to Nested Parallel Loops," *Proceedings of the 1986 International Conference on Parallel Processing,* St. Charles, IL, pp. 519-527, August, 1986.

[Poly84] C. D. Polychronopoulos, "Compiler Optimizations and Scheduling Issues for Multiprocessor Systems," Internal Memo, Office for Supercomputer Research and Development, University of Illinois, Nov., 1984.

[Poly86] C. D. Polychronopoulos, "Compiler and Hardware Issues for Fast Synchronization on Parallel Supercomputers," Technical Report in preparation, Center for Supercomputing Research and Development, University of Illinois, 1986.

[RCG 71] C. V. Ramamoorthy, K. M. Chandy and Gonzalez, "Optimal Scheduling Strategies in Multiprocessor Systems," *IEEE Trans. Compt.,* Vol. C-21, Febr., 1972.

[Rein86] S. Reinhardt, "A Data-Flow Approach to Multitasking on CRAY X-MP Computers," *Proceedings of the 10th ACM Symposium on Operating Systems Principles,* December, 1985.

[Sahn84] S. Sahni, "Scheduling Multipipeline and Multiprocessor Computers," *IEEE Trans. Compt.,* Vol. C-33, No. 7, July, 1984.

[ScGa85] K. Schwan and C. Gaimon, "Automatic Resource Allocation for the Cm* Multiprocessor," *Proc. of the 1985 International Conference on Distributed Computing Systems,* 1985.

[ShTs85] C. C. Shen and W. H. Tsai, "A graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using Minimax Criterion," *IEEE Trans. Compt.,* Vol. C-34, No. 3, March, 1985.

[Smit81] B. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *Real Time Processing IV, Proc. of SPIE,* pp. 241-248, 1981.

[Ston77] H. S. Stone, "Multiprocessor Scheduling With the Aid of Network Flow Algorithms," *IEEE Trans. on Software Engineering,* Vol. SE-3, No. 1, Jan., 1977.

[TaPe86] P. Tang and P. C. Yew, "Processor Self-Scheduling for Multiple-Nested Parallel Loops," *Proceedings of the 1986 International Conference on Parallel Processing,* August, 1986.

[Wolf82] M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," Ph.D. Thesis, University of Illinois at Urbana-Champaign, DCS Report No. UIUCCDCS-R-82-1105, 1982.

[ZhYe84] C. Q. Zhu and P. C. Yew, "A Synchronization Scheme and Its Applications for Large Multiprocessor Systems," *Proc. of the 1984 International Conference on Distributed Computing Systems,* pp. 486-493, May 1984.

[ZhYL83] C. Q. Zhu, P. C. Yew, D. H. Lawrie, "Cedar Synchronization Primitives," Laboratory for Advanced Supercomputers, Cedar Doc. No. 18, September, 1983.

VITA

Constantine D. Polychronopoulos was born on [redacted] He attended elementary and high school in Patras. After taking the national university entrance examination, he was accepted in the Department of Mathematics at the University of Athens in September 1976, where he studied theoretical and applied mathematics, and theoretical computer science. He graduated from the University of Athens in December 1980, and was awarded a Fulbright Scholarship to pursue graduate studies at Vanderbilt University, in Nashville Tennessee. He was at Vanderbilt from June 1981 until July 1982 where he received an M.S. in Computer Science. In August 1982 he joined the Department of Computer Science at the University of Illinois at Urbana-Champaign, where he worked as a Research Assistant and completed his Ph.D. under the direction of Professor David J. Kuck. Since January 1985, he has been working in the Center for Supercomputing Research and Development. After receiving his Ph.D., he will join the Center for Supercomputing Research and Development, and the Department of Electrical and Computer Engineering at the University of Illinois. Mr. Polychronopoulos is a member of the ACM and the IEEE, Computer Society.

BIBLIOGRAPHIC DATA SHEET	1. Report No. CSRD-595	2.	3. Recipient's Accession No.
4. Title and Subtitle ON PROGRAM RESTRUCTURING, SCHEDULING, AND COMMUNICATION FOR PARALLEL PROCESSOR SYSTEMS			5. Report Date August 1986 6.
7. Author(s) Constantine D. Polychronopoulos			8. Performing Organization Rept. No. CSRD-595
9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Center for Supercomputing Research and Development Urbana, IL 61801-2932			10. Project/Task/Work Unit No. 11. Contract/Grant No. US NSF DCR84-10110, US NSF DCR84-06916, US DOE-DE-FG02-85ER25001, IBM
12. Sponsoring Organization Name and Address National Science Foundation, Washington, DC U.S. Department of Energy, Washington, DC IBM Corporation, Armonk, NY Alliant Computer Corporation, Littleton, MA			13. Type of Report & Period Covered PhD Thesis 14.

15. Supplementary Notes

16. Abstracts

This dissertation discusses several software and hardware aspects of program execution on large-scale, high-performance parallel processor systems. The issues covered are program restructuring, partitioning, scheduling and interprocessor communication, synchronization, and hardware design issues of specialized units. All this work was performed focusing on a single goal: to maximize program speedup, or equivalently, to minimize parallel execution time. Parafrase, a Fortran restructuring compiler was used to transform programs in a parallel form and conduct experiments. Two new program restructuring techniques are presented, loop coalescing and subscript blocking. Compile-time and run-time scheduling schemes are covered extensively. Depending on the program construct, these algorithms generate optimal or near-optimal schedules. For the case of arbitrarily nested hybrid loops, two optimal scheduling algorithms for dynamic and static scheduling are presented. Simulation results are given for a new dynamic scheduling algorithm. The performance of this algorithm is compared to that of self-scheduling. Techniques for program partitioning and minimization of interprocessor communication for idealized program models and for real Fortran programs are also discussed. The close relationship between scheduling, interprocessor communication, and synchronization becomes apparent at several points in this work. Finally, the impact of various types of overhead on program speedup and experimental results are presented.

17. Key Words and Document Analysis. 17a. Descriptors

algorithms
architecture
compilers
operating-systems
software
synchronization

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited Distribution	UNCLASSIFIED	256
	20. Security Class (This Page) UNCLASSIFIED	22. Price