A Parallel Programming Model with Sequential Semantics

Thesis by

John Thornley

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1996
(Submitted May 20th, 1996)

Acknowledgements

Many thanks to my academic advisor, Mani Chandy, for his support, guidance, and assistance in so many ways over the previous six years. I feel very fortunate to have been advised by such a gracious and insightful academic and such an all-around nice guy. Thanks also to the other members of my examining committee, Mary Hall, Carl Kesselman, Peter Schröder, and Eric Van de Velde, for their time spent reading this thesis and for their suggestions regarding this work.

Thanks to the graduate students who have been part of our research group during my stay at Caltech, Berna Massingill, Paul Sivilotti, Adam Rifkin, Peter Hofstee, Rustan Leino, Eve Schooler, Ulla Binau, Svetlana Kryukova, Rajit Manohar, and Peter Carlin, for their help with my research and for their friendship. Particular thanks to Paul for helping me with all those "interesting little questions" (equivalence, nondeterminacy, time travel, and invisible cameras) and to Berna, Adam, and Rajit for assistance and proofreading above and beyond the call of duty. Thanks also to Diane Goodfellow for her administrative support and for keeping us well fed.

Much of this research has been influenced by two parallel programming projects at Caltech: the PCN project led by Steve Taylor and Mani Chandy at Caltech and by Ian Foster at Argonne National Labs, and the CC++ project led by Carl Kesselman and Mani Chandy at Caltech. Thanks to those individuals and their groups. Special thanks to Steve for introducing me to parallel programming with PCN during my first year at Caltech.

Thanks to the Ada 95 team at Silicon Graphics, in particular Tom Quiggle and Wes Embry, for providing me with software, computer time, and technical support for my parallel programming performance experiments. Getting my hands on a 36-processor shared-memory multiprocessor was a dream come true for me. (We computer scientists have very tame dreams!)

Thanks to my teachers and academic colleagues in the Department of Computer Science at the University of Auckland, New Zealand for providing me with so many opportunities and for encouraging me to embark upon this adventure. Thanks also to the students that I taught at the University of Auckland for shaping my perspective on computer science and for making that phase of my life so interesting and so much fun.

Thanks again to Professor Graham Hill and his team in the Department of Surgery at the University of Auckland for their compassion and skill. Thanks to Dr. Stephen Petit, Dr. Charles Bernstein, and Lucy Artinian for making me believe that bad times don't have to last forever.

Thanks to Cecilia for being my special friend and for providing me with a safe haven away from Caltech for the past five years.

Thanks to my mother, Margaret, my father, Basil, and my brother, Robert, for their unconditional love and support throughout all of my life and education. Without them, I really could not have done this!

Finally and somewhat vaguely, thanks to the California Institute of Technology for the uncountably many things that have made these last six years so fascinating and unforgettable.

This research was supported in part by Air Force Office of Scientific Research grant AFOSR-91-0070 and by the NSF under Cooperative Agreement No. CCR-9120008.

Abstract

Parallel programming is more difficult than sequential programming in part because of the complexity of reasoning, testing, and debugging in the context of concurrency. In this thesis, we present and investigate a parallel programming model that provides direct control of parallelism in a notation with sequential semantics. Our model consists of a standard sequential imperative programming notation extended with the following three pragmas:

1. The parallelizable sequence of statements pragma indicates that a sequence of statements can be executed as parallel threads.

2. The parallelizable for-loop statement pragma indicates that the iterations of a for-loop statement can be executed as parallel threads.

3. The single-assignment type pragma indicates that variables of a given type are assigned at most once and that ordinary assignment and evaluation operations can be used as implicit communication and synchronization operations between parallel threads.

In our model, a parallel program is simply an equivalent sequential program with added pragmas. The placement of the pragmas is subject to a small set of restrictions that ensure the equivalence of the parallel and sequential semantics. We prove that if standard sequential execution of a program (by ignoring the pragmas) satisfies a given specification and the pragmas are used correctly, parallel execution of the program (as directed by the pragmas) is guaranteed to satisfy the same specification.

Our model allows parallel programs to be developed using sequential reasoning, testing, and debugging techniques, prior to parallel execution for performance. Since parallelism is specified directly, sophisticated analysis and compilation techniques are not required to extract parallelism from programs. However, it is important that parallel performance issues such as granularity, load balancing, and locality be considered throughout algorithm and program development.

We describe a series of programming experiments performed on up to 32 processors of a shared-memory multiprocessor system. These experiments indicate that for a wide range of problems:

1. Our model can express sophisticated parallel algorithms with significantly less complication than traditional explicit parallel programming models.

2. Parallel programs in our model execute as efficiently as sequential programs on one processor and deliver good speedups on multiple processors.

3. Program development with our model is less difficult than with traditional explicit parallel programming models because reasoning, testing, and debugging are performed using sequential methods.

We believe that our model provides the basis of the method of choice for a large number of moderate-scale, medium-grained parallel programming applications.

Contents

List of Figures

List of Tables

List of Programs

Chapter 1

Introduction

1.1 Motivation

A parallel program is a program that is designed to execute as a group of cooperating, concurrent tasks. There are two main reasons to write parallel programs: (i) to increase performance through execution on multiprocessor computers, and (ii) to satisfy explicitly concurrent problem specifications. The scope of this thesis is parallel programming for the reason of increasing execution performance compared to equivalent sequential programs. We are not primarily concerned with explicitly concurrent problems that do not have sequential solutions, e.g., real-time monitoring and control problems.

The most direct method of parallel programming is provided by programming models with explicit parallel semantics. Constructs are provided for specifying: (i) creation and termination of parallel processes, e.g., fork-and-join, process declarations, or process spawning, and (ii) communication and synchronization between processes, e.g., message passing, locks, semaphores, monitors, or remote procedure calls. The semantics of parallel execution is usually defined to be equivalent to an interleaving of the actions of the parallel processes. This approach to parallel programming is supported by languages such as Ada [3][6] and Modula-3 [93], and by libraries such as p4 [18], Pthreads [97], PVM [119], and MPI [34][115]. Comprehensive reviews and discussions of programming models and notations with explicit parallel semantics are given by Andrews [8], Bal et al. [11], and Pancake [96].

Explicit parallel programming gives direct control of execution performance because of the close correlation between the programming model and the operation of a multiprocessor computer system. However, explicit parallel programs are often difficult to develop compared to equivalent sequential programs due to the complexity of reasoning, testing, and debugging in the context of concurrency. For example, considerable effort is often necessary to avoid race conditions, deadlock, and livelock—issues that do not arise in sequential programming. For many applications, the performance gain of explicit parallel programming does not justify the additional development costs.

An appealing alternative to explicit parallel programming is provided by automatic program parallelization [12][95]. Automatic parallelizing compilers transform sequential programs into equivalent parallel programs based on conservative data-dependence analysis. The most important techniques are transformations that allow sequential loops to be converted into parallel loops. Strengths of automatic parallelization are: (i) the programmer is shielded from most of the complexity of reasoning about concurrency, and (ii) existing sequential programs can be parallelized with relatively little effort. The major weakness is that execution performance is dependent on the compiler's success at recognizing opportunities for efficient parallel execution.

Automatic program parallelization has been shown to be fairly successful at detecting loop-level parallelism in scientific programs that operate on matrices. However, more general applicability has not been demonstrated convincingly. The essential difficulty is that an efficient sequential algorithm is not necessarily a good basis for an efficient parallel algorithm. For example, quicksort is widely accepted as the most efficient general-purpose sequential sorting algorithm [98, section 8.2][107, Chapter 9], yet the parallelism in quicksort does not scale to more than a few processors [29][36]. Usually, there is too little information in the text of a sequential program to allow automatic transformation into an equivalent parallel program with entirely different algorithms and data structures.

In this thesis, we present and investigate a parallel programming model that combines the advantages of both explicit parallel programming and automatic program parallelization. The programming model consists of a standard sequential notation extended with a small set of pragmas. These pragmas are used to indicate where statements and loop iterations can be executed in parallel and where assignment and evaluation of variables can be used as implicit communication and synchronization operations between parallel threads. If the pragmas are used correctly, parallel execution of a program as directed by the pragmas is equivalent to standard sequential execution. The key idea is direct control of parallelism in a programming model with sequential semantics.

This model allows parallel programs to be developed with sequential reasoning, testing, and debugging, prior to parallel execution for performance. Sequential reasoning is performed by ignoring the pragmas, except to verify their correct use. Sequential testing and debugging are performed by instructing the compiler to disregard the pragmas and generate ordinary sequential code. Parallel execution is achieved by instructing the compiler to generate parallel code as indicated by the pragmas. Since parallelism is specified directly, sophisticated analysis and compilation techniques are not required. If the sequential interpretation of a program satisfies a given specification and the pragmas are used correctly, the parallel interpretation of the program is guaranteed to satisfy the same specification.

Although this programming model removes the difficulty of parallel reasoning, testing, and debugging, it does not lessen the need to explicitly consider parallel performance issues during program design and development. The methodology that we propose is not to first develop a sequential program without regard to parallelism, then later add pragmas. Such an approach would usually lead to an inefficient parallel program, for the same reasons that limit the applicability of automatic program parallelization. The appropriate methodology is to develop a parallel program from the outset, but to do so using sequential reasoning, testing, and debugging techniques. Parallel performance issues, e.g., granularity, load balancing, and locality, should be taken into account throughout the process of program development.

The goal of this work is not to develop a more efficient parallel programming model than other models with explicit parallel semantics. Rather, it is to reduce the difficulty of parallel programming, without significant sacrifice of efficiency, through the use of sequential programming methods. Our interest in parallel programming is not solely focused on the traditional arena of scientific computation on parallel supercomputers, where performance is often of paramount importance. We believe that the proliferation of small-scale to moderate-scale multiprocessor servers, workstations, and personal computers will also lead to a greatly expanded role for parallel programming in a wide range of commodity and specialized software. For many of these applications, parallel programming will be feasible only if program development and maintenance costs are not significantly greater than those of sequential programming.

1.2 Programming Model

The parallel programming model that we investigate consists of a structured, imperative, sequential programming notation extended with the following three pragmas:

1. The parallelizable sequence of statements pragma is applied to a sequence of statements to indicate that the statements can be executed as parallel threads.

2. The parallelizable for-loop statement pragma is applied to a for-loop statement to indicate that the iterations of the loop can be executed as parallel threads.

3. The single-assignment type pragma is applied to a type declaration to indicate that assignment and evaluation of variables of that type can be used as communication and synchronization operations between parallel threads. Types that are not single-assignment types are referred to as mutable types.

Additional support pragmas are provided to control thread scheduling priorities and parallel error checking. Part of the model is a set of restrictions on the use of the pragmas that ensures the equivalence of the parallel and sequential interpretations of the constructs. The fundamental restrictions on the placement of the pragmas in a sequential program are as follows:

- It is illegal to jump into, out of, or between the statements of a parallelizable sequence of statements or parallelizable for-loop statement.

- It is an error to evaluate an unassigned single-assignment variable or to assign a previously assigned single-assignment variable.

- It is an error for one statement or iteration to assign a mutable variable that is assigned or evaluated by another statement or iteration of the same parallelizable sequence of statements or parallelizable for-loop statement, unless those actions are synchronized.

- It is an error to depend on a particular kind of exception being propagated out of a parallelizable sequence of statements or parallelizable for-loop statement.

Some of these restrictions can be checked at compile time, some can be checked with low overhead at run time, and some can either be left the programmer to verify or be checked with high overhead at run time.

If the pragmas are used correctly, execution of the constructs according to the parallel semantics is equivalent to execution according to the standard sequential semantics. The parallel semantics is based on the following framework:

1. The statements of a parallelizable sequence of statements are executed as parallel threads. Execution of the parallelizable sequence of statements terminates when all the statements have terminated.

2. The iterations of a parallelizable for-loop statement are executed as parallel threads. Execution of the parallelizable for-loop statement terminates when all the iterations have terminated.

3. Evaluation of an unassigned single-assignment variable causes the evaluating thread to suspend until the variable is assigned by another parallel thread.

In this thesis, we prove the equivalence of the parallel and sequential semantics of our programming model, for programs without errors. Therefore, if the sequential interpretation of a program satisfies a given specification, the parallel interpretation of the program satisfies the same specification.

1.3 Programming Methodology

The parallel programming methodology that we explore is one in which parallel performance is considered throughout program development, but reasoning, testing, and debugging are performed using sequential techniques. The sequential semantics provides the basis for reasoning about correctness, testing, and debugging. The parallel semantics provides the basis for parallel performance design and analysis.

A parallel program can be shown to satisfy its specification by showing that the sequential program without the pragmas satisfies the specification and that the pragmas do not violate any of the restrictions. Program development, testing, and debugging can be performed sequentially, by disregarding the pragmas and compiling the program into sequential code. Standard compilers and debugging tools can be used for this sequential phase of program development. After testing and debugging, a program can be compiled into parallel code as indicated by the pragmas and executed on a multiprocessor computer for parallel performance. Parallel execution will produce the same results as sequential execution, except for possibly different behavior for some parallel error conditions.

The efficiency of a parallel program is under the direct control of the programmer. From the outset of program development, the programmer should consider the parallel performance consequences of algorithm and program design choices for the intended target architecture. The major issues affecting parallel performance are granularity, load balancing, and locality of data accesses. Simply adding the pragmas to a existing sequential program is unlikely to result in an efficient parallel program. An efficient parallel program will usually require to some degree different algorithms and data structures than an efficient sequential program. However, a sequential program will often be a suitable starting point for the development of an efficient parallel program.

1.4 A Simple Example Program

In this section, we present a small program as an introductory example of the parallel programming model and methodology that we investigate in this thesis. The LU_Factorize procedure in Program 1.1 computes the unit lower triangular and upper triangular factors of an input matrix, A, using Crout's method without pivoting [98, Section 2.3]. The computed factors are overlaid in the output matrix, LU. An example is shown in Figure 1.1. In the interest of brevity, the procedure specification ignores the imprecision of floating-point arithmetic and the possibility of division by zero due to the absence of pivoting.

$$\underset{\mathbf{A}}{\begin{bmatrix} 6 & 3 & 5 & 9 & 4 & 7 & 8 & 3 & 4 \\ 18 & 13 & 18 & 23 & 9 & 22 & 28 & 11 & 9 \\ 42 & 37 & 45 & 52 & 23 & 50 & 71 & 37 & 20 \\ 24 & 4 & 2 & 81 & 68 & 15 & 23 & 58 & 54 \\ 18 & 37 & 32 & 65 & 43 & 82 & 91 & 58 & 67 \\ 12 & 26 & 31 & 11 & 18 & 81 & 70 & 24 & 40 \\ 36 & 30 & 25 & 56 & 94 & 11 & 25 & 62 & 18 \\ 18 & 29 & 38 & 29 & 5 & 78 & 80 & 2 & 24 \\ 42 & 37 & 37 & 86 & 41 & 50 & 94 & 64 & 39 \end{bmatrix}} = \underset{\mathbf{L}}{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & -2 & 6 & 1 & 0 & 0 & 0 & 0 & 0 \\ 3 & 7 & 2 & 8 & 1 & 0 & 0 & 0 & 0 \\ 2 & 5 & -3 & 4 & 5 & 1 & 0 & 0 & 0 \\ 6 & 3 & 7 & -3 & 7 & -4 & 1 & 0 & 0 \\ 3 & 5 & -4 & 6 & 2 & -2 & 3 & 1 & 0 \\ 7 & 4 & 5 & 2 & -3 & 2 & 2 & -4 & 1 \end{bmatrix}} \times \underset{\mathbf{U}}{\begin{bmatrix} 6 & 3 & 5 & 9 & 4 & 7 & 8 & 3 & 4 \\ 0 & 4 & 3 & -4 & -3 & 1 & 4 & 2 & -3 \\ 0 & 0 & -2 & 5 & 7 & -3 & -1 & 8 & 4 \\ 0 & 0 & 0 & 7 & 4 & 7 & 5 & 2 & 8 \\ 0 & 0 & 0 & 0 & 6 & 4 & 1 & 3 & 4 \\ 0 & 0 & 0 & 0 & 0 & 5 & 6 & 9 & 7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \end{bmatrix}}$$

Figure 1.1: An example of LU factorization.

The program is derived from the following formula, which defines LU(I, J) in terms of A(I, J) and other components of LU:

$$\mathrm{LU}_{i,j} = \begin{cases} (\mathrm{A}_{i,j} - \sum_{k=1}^{j-1} \mathrm{LU}_{i,k} \times \mathrm{LU}_{k,j})/\mathrm{LU}_{j,j} & \text{for } 1 \le i \le \mathrm{N}, 1 \le j < i \\ \mathrm{A}_{i,j} - \sum_{k=1}^{i-1} \mathrm{LU}_{i,k} \times \mathrm{LU}_{k,j} & \text{for } 1 \le i \le \mathrm{N}, i \le j \le \mathrm{N} \end{cases}$$

Computation of LU(I, J) is dependent on the prior computation of components of LU above and to the left of LU(I, J), as shown in Figure 1.2(a). Ignoring the pragmas, the program is a sequential program in which the components of LU are computed in order of increasing rows and columns, as shown in Figure 1.2(b). The pragmas indicate how the program can be executed as a parallel program in which the components of LU are computed concurrently. Computation of each component will automatically suspend when it attempts to evaluate an unassigned component and resume execution when that component is assigned. In this manner, the data dependencies implicitly control the order of computation. An example of parallel execution is shown in Figure 1.2(c).

```
type Single_Float is new Float;
pragma Single_Assignment(Single_Float);
type Matrix is array (1 .. N, 1 .. N) of Single_Float;

procedure LU_Factorize (A : in Matrix; LU : out Matrix) is
--| requires
--|       Nonsingular(A).
--| ensures
--|       Unit_Lower_Triangle(LU)*Upper_Triangle(LU) = A.
begin
    pragma Parallelizable_Loop;
    for I in 1 .. N loop
        pragma Parallelizable_Sequence;
        pragma Parallelizable_Loop;
        for J in 1 .. I - 1 loop
            declare
                Sum : Float := 0.0;
            begin
                for K in 1 .. J - 1 loop
                    Sum := Sum + Float(LU(I, K)*LU(K, J));
                end loop;
                LU(I, J) := (A(I, J) - Single_Float(Sum))/LU(J, J);
            end;
        end loop;
        pragma Parallelizable_Loop;
        for J in I .. N loop
            declare
                Sum : Float := 0.0;
            begin
                for K in 1 .. I - 1 loop
                    Sum := Sum + Float(LU(I, K)*LU(K, J));
                end loop;
                LU(I, J) := A(I, J) - Single_Float(Sum);
            end;
        end loop;
    end loop;
end LU_Factorize;
```

Program 1.1: LU factorization.

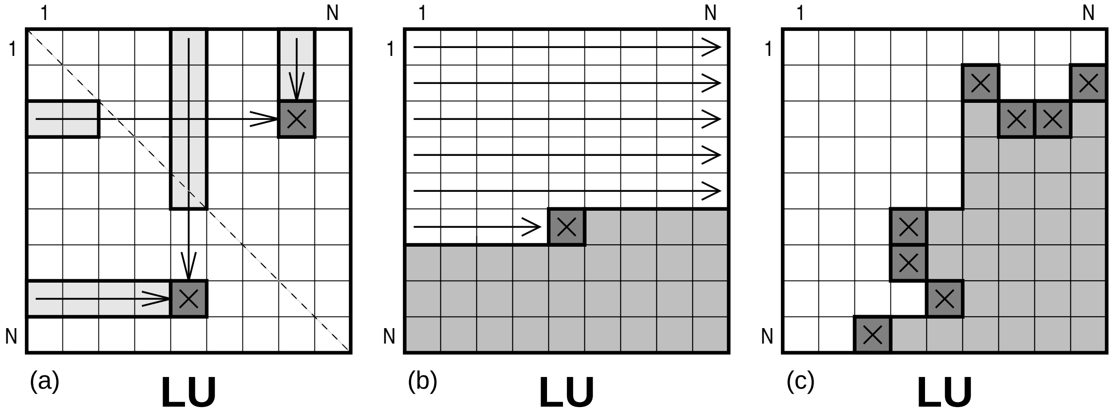


Figure 1.2: LU factorization: (a) Data dependencies in computation of LU(I, J). (b) Snapshot of sequential execution. (c) Possible snapshot of parallel execution.

The Single_Assignment pragma on line 2 indicates that all variables of type Single_Float are assigned at most once. The Parallelizable_Loop and Parallelizable_Sequence pragmas on lines 12, 14, 15, and 26 indicate that the N^2 executions of the inner blocks can be executed as parallel threads. Each parallel thread assigns to one LU(I, J) and evaluates other components of LU. Parallel assignment and evaluation operations on the components of LU are permitted because the components are single-assignment variables. These operations are implicit synchronization points between the parallel threads. There are no parallel operations on mutable variables.

In parallel execution of the program, the order of execution of the parallel threads is in part determined by the data dependencies. However, there are many different execution orderings that satisfy the data dependencies. For example, on a single processor, parallel execution may compute the components in the same order as sequential execution, whereas on multiple processors, many components may be computed truly concurrently and the order of computation may be nondeterministic. The correctness of all parallel execution orderings is guaranteed by the correctness of the sequential interpretation of the program and the fact that the pragmas do not violate any of the restrictions.

This simple program is not intended as a demonstration of an efficient parallel program. In practice, the benefits of parallel execution would be overwhelmed by the costs of thread management and synchronization operations. In Chapter 7, we present an efficient variation of this program in which the use of parallelism and single-assignment variables is more coarse grained.

1.5 Key Questions

The equivalence of the parallel and sequential semantics of our programming model is achieved at the cost of the restrictions on the programs that are allowed by the model. There are two ways to view the restrictions: (i) sequential programs are restricted to those that have the same meaning when interpreted as parallel programs, or (ii) parallel programs are restricted to those that have the same meaning when interpreted as sequential programs. The essential restriction on the sequential interpretation of a program is the limitation on access to mutable variables in parallelizable constructs. The essential restriction on the parallel interpretation of a program is the limitation on nondeterminacy as a result of using single-assignment variables as the only means of communication and synchronization between parallel threads.

The practical value of our programming model and methodology is determined by the answers to the following two key questions:

1. Expressiveness: How significantly do the restrictions limit the algorithms and data structures that we can express?

2. Efficiency: How significantly do the restrictions limit the efficiency of the programs that we can write?

In this thesis, we describe programming experiments designed to investigate these questions. We present parallel programs to solve a range of problems that require a variety of algorithmic patterns and data structures. For some of these programs, we analyze performance measurements gathered on up to 32 processors of a symmetric multiprocessor computer system. From these programming experiments, we draw conclusions regarding the strengths and weaknesses of our programming model and methodology.

1.6 Key Findings

Our programming experiments indicate that for a wide range of problems our programming model is able to express parallel programs that have sophisticated control and synchronization patterns, yet are: (i) not much more complicated than efficient sequential programs developed without regard to parallelism, and (ii) significantly less complicated than efficient parallel programs expressed using traditional explicit parallel programming models, e.g., thread libraries with barrier and lock synchronization.

We found that the integration of single-assignment types with the type system of a traditional sequential programming notation provides a powerful means of expressing synchronization based on data flow at a high level of abstraction. The integration of single-assignment types and mutable types within the same type system provides concise control of the granularity of synchronization through the inclusion of single-assignment components in otherwise mutable data structures. Parallelizable for-loop statements with arguments provide control of the granularity of parallelism without adding additional complexity to the structure of the program.

Our performance experiments indicate that our programming model is able to express moderate-scale, medium-grained parallel programs that: (i) execute as efficiently as efficient sequential programs on one processor, (ii) deliver good speedups on multiple processors, and (iii) transparently adapt to dynamically changing processing resources. This is exactly the behavior that is required for parallel programming to make an impact in mainstream computing. The performance measurements also indicate that the use of single-assignment types as a synchronization mechanism is as efficient as the use of less-structured constructs such as locks.

Our experience in developing the experimental programs confirms the benefits of a parallel programming model with equivalent sequential semantics. All of our programs were developed, tested, and debugged entirely sequentially, and no errors were discovered in the subsequent parallel execution. We found that errors relating to the restrictions on the pragmas were easily avoided, since subtle cases rarely occur in practice. Development of equivalent programs using traditional explicit parallel programming models would have been considerably more difficult and error prone. We believe that our model provides the basis of the method of choice for a large number of moderate-scale, medium-grained parallel programming applications.

1.7 Thesis Outline

In Chapter 2, we summarize the background of the parallel programming model that we explore in this thesis. The pragmas in our model are based on constructs that date back to the 1960s in the context of other parallel programming notations.

In Chapter 3, we present our parallel programming model as a small set of pragmas added to a standard sequential notation. We describe the restrictions on the use of the pragmas, and we define a parallel semantics for the pragmas. In Chapter 4, we prove that the parallel semantics is equivalent to the standard sequential semantics of the programming notation, if the pragmas are used correctly. This equivalence result forms the basis of our parallel programming methodology.

In Chapter 5, we discuss the issues that affect the performance of programs in our parallel programming model on a shared-memory multiprocessor. We give measurements of the execution costs of the fundamental operations in the parallel programming model. In Chapters 6 and 7, we describe parallel programming experiments designed to investigate general programming and performance issues. We give performance measurements for some of the experimental programs on a shared-memory multiprocessor.

In Chapter 8, we consider the consequences of the limitations on nondeterminacy in our programming model. We describe a class of problems where increased nondeterminacy allows more efficient parallel algorithms. As a result, we investigate the integration of nondeterministic constructs with our programming model.

In Chapter 9, we discuss additional implementation, programming, and performance issues for our parallel programming model on distributed-memory computer systems.

In Chapter 10, we compare our programming model and methodology with related work that is also motivated by the goal of reducing the difficulty of reasoning about explicit parallelism. In particular, we contrast our approach with automatic parallelizing compilers, run-time parallelization systems, parallel declarative programming, and data-parallel programming.

In Chapter 11, we conclude with an assessment of our parallel programming methodology and we suggest directions for future development.

Chapter 2

Background

The parallel programming model that we investigate in this thesis consists of a sequential imperative language extended with pragmas for parallelizable sequences of statements, parallelizable for-loop statements, and single-assignment types. These pragmas are based on constructs that date back to the 1960s in the context of other parallel programming notations. In this chapter, we review the background of the individual constructs and of their integration with sequential imperative programming notations. We then compare our programming model with other parallel programming models that incorporate these constructs. In Chapter 10, we compare our programming model to other approaches to reducing the difficulty of reasoning about explicit parallelism.

2.1 Parallel Composition of Statements

Our parallelizable sequence of statements pragma is based on the parallel composition of statements construct. In 1966, Wirth [135] pointed out the difference between: (i) parallelism for multiprocessor performance, and (ii) parallelism to represent concurrency in the problem specification. For the first case, he advocated a parallel composition of statements construct based on the syntax of standard sequential composition of statements, as opposed to providing an entirely different notation for parallel execution, such as fork-and-join. Wirth suggested the use of **and** in place of ";" between statements. In 1968, Dijkstra [33] proposed the **parbegin**–**parend** notation for parallel composition of statements.

The first major language to support parallel composition of statements was Algol 68 [133], which incorporated the **parbegin**–**parend** notation. Other early notations to support parallel composition of statements include CSP [60][61] and Occam [86] (a programming language derived from CSP). Many subsequent parallel programming notations express parallel execution using some form of parallel composition of statements.

2.2 Parallel For-Loop Statement

Our parallelizable for-loop statement pragma is based on a general parallel for-loop statement. This form of parallel for-loop statement is simply a quantified parallel composition of statements. In his 1968 paper, Dijkstra discussed the need for quantified parallel composition of statements. He used an informal notation that was not based on the syntax of the sequential for-loop statement.

The initial version of CSP incorporated a limited form of quantified parallel composition. However, it was not integrated with sequential repetition. Occam is the earliest implemented language that we know of to integrate quantified parallel composition of statements with the syntax for sequential iteration. The Occam replicator construct can be either sequential or parallel, depending on whether the **seq** or **par** keyword is used.

Many data-parallel notations [58] incorporate more restricted (usually synchronous) forms of parallel loops, such as the `FORALL` construct [4] in some Fortran dialects. The `INDEPENDENT` directive of HPF (High Performance Fortran) [40][74] is an assertion that the iterations of a loop are independent and can be executed in parallel. Unlike our parallelizable for-loop statement pragma, no interaction is permitted between the iterations.

2.3 Single-Assignment Variables

In 1968, Tesler and Enea [123] described the use of single-assignment variables as a sequencing mechanism in their parallel programming notation, Compel. In Compel, the single-assignment restriction enables automatic compile-time scheduling of the concurrent execution of statements.

Since the mid-1970s, single-assignment variables have been used for run-time synchronization in parallel dataflow languages such as Id [9][94], Val [1][88], and Sisal [38][87]. A review of the principles and early development of parallel dataflow programming is given by Ackerman [2]. Since the early 1980s, single-assignment variables have been used for run-time synchronization in parallel logic programming languages such as Concurrent Prolog [109][110], Parlog [27][28], and Strand [43]. A review of the principles and history of parallel logic programming is given by Shapiro [113]. Shapiro also edited a collection of seminal papers on parallel logic programming [111][112].

Single-assignment variables are in many ways similar to the concept of futures incorporated in some parallel functional programming notations. In the mid-1980s, futures were incorporated in parallel functional programming languages such as Multilisp [54] and Qlisp [50]. A collection of papers relating to parallel functional programming using futures was edited by Ito and Halstead [68].

2.4 Integration with Sequential Imperative Programming

In 1977, Kessels [71] described a conceptual framework that integrated single-assignment and mutable types, parallel and sequential composition of statements, and parallel and sequential for-loop statements. However, we are not aware of this framework directly leading to the design and implementation of any actual programming language.

Designed in the late 1980s, PCN [23][24][42] is the first implemented language that we know of to incorporate single-assignment and mutable types and parallel and sequential composition of statements. A formal operational semantics and proof rules have been developed for PCN [24, Part III]. Much of the structure of PCN is derived from parallel logic programming. In particular, neither loops nor functions are supported, and single-assignment and mutable types belong to separate type systems with completely different compatibility rules.

Declarative Ada [124][127] is a notation designed and implemented (in earlier work by the author of this thesis) to experiment with the integration of some key ideas from PCN with a traditional sequential imperative language. Single-assignment types, parallel composition of statements, and a parallel for-loop statement are integrated with the types and statements of a small sequential subset of Ada [6]. Parallel Declarative Ada programs can be developed to be identical to equivalent sequential Ada programs.

A recent parallel programming language, CC++ [21][22], integrates single-assignment types, parallel composition of statements, a parallel for-loop statement, and other extensions with the full C++ [117] language. CC++ supports many styles of parallel programming, including the parallel programming methodology that we explore in this thesis.

2.5 Comparison with Our Model

The significant difference between the programming model that we define and investigate in this thesis and the notations described in Section 2.4 is that our programming model is restricted so that it can be defined with a sequential semantics. In the other notations, parallel composition of statements, parallel for-loop statements, and single-assignment types are more general and can only be defined with a parallel semantics. A parallel semantics is defined in terms of concepts such as concurrency, interleaving, and suspension.

For example, consider the following two parallelizable sequences of statements in our parallel programming notation:

```
begin                                   begin
    pragma Parallelizable_Sequence;         pragma Parallelizable_Sequence;
    X := 5;                                 Y := X;
    Y := X;                                 X := 5;
end;                                    end;
```

Assume that the variables X and Y are single-assignment integer variables. The sequence of statements on the left is allowed, but execution of the sequence of statements on the right is erroneous, because X is evaluated before it is assigned in the sequential interpretation. In contrast, consider the following two analogous CC++ parallel blocks:

```
par {                                   par {
    x = 5;                                  y = x;
    y = x;                                  x = 5;
}                                       }
```

Assume that the variables x and y are single-assignment integer variables. Both blocks are allowed and have exactly the same meaning, because the order of the individual statements in a CC++ parallel block is not significant.

The sequential semantics of our programming model defines the pragmas to have no effect whatsoever on the standard sequential meaning of the program. There are four important benefits of having a sequential semantics:

1. Reasoning about correctness: The programmer can reason about the correctness of parallel programs without reasoning about concurrency.

2. Parallelizable for-loop statement variations: The iterations of a parallelizable for-loop can be assigned to a smaller number of parallel threads in a blocked, cyclic, or on-demand pattern to reduce thread creation and scheduling costs. For the more general parallel for-loop statements of Declarative Ada and CC++, this might result in deadlock.

3. Optimizing compilation: The compiler can translate any parallelizable construct into sequential code for efficiency. In general, with PCN, Declarative Ada, and CC++, it is impossible to determine whether the sequential interpretation of a parallel construct is equivalent to the parallel interpretation.

4. Nonpreemptive scheduling: Run-time scheduling of parallel threads can be nonpreemptive, thereby reducing thread scheduling and cache invalidation costs. Since PCN, Declarative Ada, and CC++ programs may include parallel composition of nonterminating computations, the thread scheduling strategy is required to be preemptive.

These benefits arise at the cost of restricting the applicability of our parallel programming model to problems that have sequential solutions.

Chapter 3

A Programming Model with Parallel and Sequential Semantics

In this chapter, we present the parallel programming model that we explore in this thesis, as a small set of pragmas added to a standard sequential notation. The pragmas are used to indicate where the execution of constructs according to a parallel semantics is equivalent to their execution according to the standard sequential semantics. Part of the model is a set of restrictions on the use of the pragmas that ensures the equivalence of the parallel and sequential semantics. For each of the pragmas, we: (i) give the syntax of the pragma, (ii) summarize the standard sequential semantics of the construct to which the pragma is applied, (iii) describe the restrictions on the use of the pragma, and (iv) define the equivalent parallel semantics associated with the pragma. A proof of the equivalence of the parallel and sequential semantics of our programming model is given in Chapter 4.

3.1 Framework

3.1.1 The Underlying Language: Ada

Our parallel programming model could be integrated with almost any structured imperative programming language. The major requirement is that the underlying language provide structured choice and iteration constructs, rather than only goto statements. It is also useful for the language to support dynamic memory allocation. Suitable languages include Pascal, C, Ada, Modula-3, C++, and Fortran 90. We choose to define our parallel programming model as a small set of pragmas added to Ada [3][6]. To avoid complicating the definition with too many Ada-specific details, we define the model in the context of Ada without tasking. In Chapter 8, we consider the integration of our model with the standard Ada tasking model.

Ada is an established and internationally standardized programming language that supports constructs typical of modern imperative languages. The original Ada language, Ada 83 [6], incorporates: (i) integer, floating-point, fixed-point, enumeration, character, boolean, array, record, and access (i.e., pointer) data types, (ii) assignment, block, if-then-else, case, while-loop, for-loop, loop-exit, function-return, and procedure-call statements, (iii) procedures and functions with **in**, **in out**, and **out** mode parameters, (iv) subprogram and operator overloading, (v) packages for data abstraction, (vi) separate compilation of packages and subprograms, (vii) concurrent tasks with communication and synchronization constructs, (viii) exception handling, (ix) generic subprograms and packages, and (x) standardized input-output libraries.

The updated Ada language, Ada 95 [3], is a superset of Ada 83 that adds: (xi) additional support for object-oriented programming, (xii) relaxation of some restrictions on access types, (xiii) support for extensible and hierarchical library packages, (xiv) additional communication and synchronization constructs, (xv) additional standardized libraries, and (xvi) additional support for systems programming, real-time systems, distributed systems, information systems, numeric programming, safety and security, and interfacing to other languages. In the interest of general readability, we restrict our example programs to a subset of Ada features. However, our programming model is compatible with the entire Ada language.

3.1.2 Why Ada?

There are a number of reasons why we choose Ada as the underlying language for the presentation of our parallel programming model.

Practical Issues: (i) Ada is stable, well-defined, and internationally (ISO and ANSI) standardized. (ii) Ada has a verbose syntax that makes programs easy to read and understand by anyone familiar with languages such as Pascal, C, or Fortran. (iii) Validated Ada compilers are available for almost all computer architectures and operating systems. (iv) Ada has a well-defined syntax for pragmas, and Ada compilers are required to simply ignore any pragmas that they do not recognize. Therefore, we can test our programs sequentially using standard Ada compilers.

Design Issues: (i) Ada has a well-defined hierarchy of compile-time errors, run-time errors, and run-time exceptions. This framework is useful in clearly describing program errors in our model. (ii) The loop parameter of an Ada for-loop statement is implicitly declared and cannot be modified by the execution of the loop body. We require this of our parallelizable for-loop statement. (iii) An Ada goto statement cannot jump from outside to inside a sequence of statements. We require this for our parallelizable sequence of statements and parallelizable for-loop statement. (iv) The default initial value of a scalar variable is undefined. We require this of our single-assignment variables.

Tasking Issues: (i) Since Ada supports tasking, memory allocation and deallocation operations are required to be "thread-safe", i.e., they are required to be able to be executed in parallel without interference. This is not necessarily true of the memory allocation and deallocation operations provided by a purely sequential language. (ii) The Ada task priority model can be extended to allow priorities to be assigned to the statements of our parallelizable sequence of statements and the iterations of our parallelizable for-loop statement. Assigning priorities does not affect the semantics of the pragmas, but may improve run-time performance. (iii) Our parallel programming model can be integrated with the Ada tasking model for problems that require some nondeterministic behavior. (iv) The parallel semantics of our programming model can be implemented by straightforward transformation of our parallelizable constructs into standard Ada tasking constructs. This provides us with a simple method of testing and measuring the parallel performance of our programs.

3.1.3 Errors and Exceptions

In our programming model, as in Ada, errors are classified as either illegal constructs, exceptional execution situations, or erroneous execution situations.

Illegal Constructs: An illegal construct can and must be detected and reported by the compiler, linker, or loader. Therefore, an illegal program can never be executed. For example, it is illegal to assign an integer-valued expression to a floating-point variable without an explicit type conversion. Our programming model defines seven illegal uses of the pragmas, which we label I.1 through I.7 for future reference.

Exceptional Execution: An exceptional execution situation always raises a specified predefined exception. The exception propagates to the outermost program or task execution, unless it is explicitly handled. For example, the predefined Constraint_Error exception is raised if an array index is out of range. Our programming model does not define any new exceptional execution situations. However, an implementation may choose to handle some of the new erroneous execution situations by raising an exception, as described in Section 3.8.

Erroneous Execution: The behavior of an erroneous execution situation is undefined. The program may terminate abnormally, continue execution, raise an exception, or suspend without termination. For example, the execution of a program is erroneous if it attempts to evaluate an undefined scalar variable. Our programming model defines six new erroneous execution situations, which we label E.1 through E.6 for future reference.

3.1.4 Sequential and Parallel Semantics

The semantics of a programming notation is a definition of the meaning of all the legal programs allowed by the notation. An "operational semantics" defines the meaning of programs in terms of their interpretation by an abstract machine. Throughout this thesis, we use the term "sequential semantics" to mean an operational semantics in which the abstract machine maintains a single thread of control in the interpreted program, and the term "parallel semantics" to mean an operational semantics in which the abstract machine may maintain multiple threads of control in the interpreted program. For example, a conventional operational semantics for a sequential imperative language, such as Fortran, Pascal, C, C++, or Ada without tasking, would be a sequential semantics. A conventional operational semantics for a multithreaded notation, such as CSP, Ada tasking, or a thread library (e.g., Pthreads), would be a parallel semantics.

3.1.5 Method of Definition

We define the syntax of our pragmas using the same notation that is used in the Ada language definition [3][6]. In this notation, items enclosed in square brackets are optional. In all other respects, the syntax of the pragmas is trivial.

We define the sequential and parallel semantics of our programming model using an informal written description rather than a formal definition using a mathematical model. This is the approach taken in the primary definitions of almost all popular programming notations, including Ada. We use the same terminology as that used in the Ada language definition. In Chapter 4, where we prove the equivalence of the parallel semantics and sequential semantics, we are more precise about the semantics of parallel execution.

The intention of this thesis is not to provide a language definition document for a parallel extension of Ada. Ada is simply a convenient basis for describing our general parallel programming model and methodology. Therefore, we do not explicitly discuss the implications of the interaction of our programming model with every Ada-specific feature. To do so would be tedious for most of our readers and add nothing to achieving the purpose of this thesis.

3.1.6 Definition of Restrictions

The pragmas in our parallel programming model are subject to a set of restrictions chosen to ensure that execution according to the parallel semantics is equivalent to execution according to the standard sequential semantics. Most of the restrictions are defined in the context of the sequential semantics. However, in Section 3.5 and Section 3.6 we describe two restrictions that must be defined in the context of the parallel semantics. This is unfortunate because it means that reasoning in the context of the parallel semantics is required to show that these restrictions are satisfied. Fortunately, there are slightly stronger versions of the two restrictions that can be defined in the context of the sequential semantics. Therefore, in most cases, sequential reasoning can be used to show that the pragmas satisfy the restrictions.

3.1.7 Memory Model

The model that we define is a shared-memory parallel programming model. All parallel threads share access to a single logical address space. The same kind of memory model is provided by the standard Ada tasking model and by thread libraries such as Pthreads. The programming model may be implemented on top of a system in which all processors genuinely share uniform access to a single memory system. Alternatively, the shared-memory programming model may be implemented as a layer on top of a physically distributed memory hierarchy with non-uniform memory access mechanisms. In this thesis, we discuss the performance issues associated with both implementation platforms.

3.2 The Parallelizable Sequence of Statements Pragma

3.2.1 Syntax

A Parallelizable_Sequence pragma is applied to a sequence of statements to indicate that the statements can be executed as parallel threads. The form of a Parallelizable_Sequence pragma is as follows:

```
pragma Parallelizable_Sequence;
statement
...
statement
```

A Parallelizable_Sequence pragma is allowed only immediately preceding a sequence of statements. The sequence of statements is referred to as a parallelizable sequence of statements.

3.2.2 Summary of the Standard Sequential Semantics

The sequential semantics of a parallelizable sequence of statements is the same as the standard sequential semantics of a sequence of statements. Execution of a sequence of statements consists of the execution of the individual statements in succession until the execution of all the statements is complete or a transfer of control out of the sequence of statements takes place. It is illegal for a goto statement to transfer control into a sequence of statements.

Transfer of control out of a sequence of statements can be caused by: (i) execution of an exit statement, return statement, or goto statement within the sequence of statements, or (ii) the raising of an exception within the sequence of statements. Because of the restrictions described in the Section 3.2.3, a transfer of control out of a parallelizable sequence of statements can only be caused by the raising of an exception within the sequence of statements.

3.2.3 Restrictions

In order for the parallel semantics to be equivalent to the standard sequential semantics, a parallelizable sequence of statements is subject to the standard restrictions on a sequence of statements, plus a small number of additional restrictions, described below. These additional restrictions are the same as or analogous to the additional restrictions on a parallelizable for-loop statement, described in Section 3.3.3.

Restrictions on Transfer of Control:

- It is illegal for an exit statement, return statement, or goto statement to transfer control out of a parallelizable sequence of statements. (I.1)

- It is illegal for a goto statement to transfer control between the statements of a parallelizable sequence of statements. (I.2)

Restrictions on Shared Variables:

- It is erroneous for one statement of a parallelizable sequence of statements to assign a mutable variable that is assigned or evaluated by another statement of the same parallelizable sequence of statements, unless those actions are synchronized. (E.1)

- Similarly, it is erroneous for one statement of a parallelizable sequence of statements to deallocate any dynamically-allocated variable that is assigned or evaluated by another statement of the same parallelizable sequence of statements, unless those actions are synchronized. (E.2)

These two restrictions are described in more detail in Section 3.5.

Restrictions on Exception Handling:

- It is erroneous to make a choice based on the kind of exception that is propagated out of a parallelizable sequence of statements that terminates exceptionally. (E.3)

- It is erroneous to evaluate or assign certain variables after a parallelizable sequence of statements that terminates exceptionally. This restriction is described in more detail in Section 3.6. (E.4)

3.2.4 Equivalent Parallel Semantics

A parallelizable sequence of statements has the following parallel semantics:

- Initiation of the execution of a parallelizable sequence of statements consists of the initiation of the execution of each statement as a separate parallel thread of control. The order in which the execution of the statements is initiated is undefined.

- If execution of all the statements terminates normally, execution of the parallelizable sequence of statements terminates normally.

- If execution of one of the statements terminates exceptionally, execution of the other statements is aborted and execution of the parallelizable sequence of statements terminates exceptionally. The kind of exception that is propagated is undefined.

If all of the restrictions are satisfied, execution of a parallelizable sequence of statements according to this parallel semantics is equivalent to execution according to the standard sequential semantics.

3.2.5 Examples

The following is a simple example of a parallelizable sequence of statements:

```
begin
    pragma Parallelizable_Sequence;
    X := 1;
    Y := 2;
end;
```

The following function is illegal (violates restriction I.1), because the return statement transfers control out of the parallelizable sequence of statements:

```
function Zero_Vector return Vector is
    Result : Vector;
begin
    pragma Parallelizable_Sequence;
    Result.X := 0;
    Result.Y := 0;
    Result.Z := 0;
    return Result;
end Zero_Vector;
```

If X and Y are mutable variables, execution of the following parallelizable sequence of statements is erroneous (violates restriction E.1), because X is assigned by one statement and evaluated by another statement:

```
begin
    pragma Parallelizable_Sequence;
    X := 1;
    Y := X;
end;
```

Execution of the following block statement is erroneous (violates restriction E.3), because the exception handler makes a choice based on the kind of exception that is propagated out of the parallelizable sequence of statements:

```
begin
    pragma Parallelizable_Sequence;
    A(I) := B(I)*C(I);
    A(J) := B(J)*C(J);
exception
    when Numeric_Error => Put_Line("Overflow");
end;
```

In contrast, execution of the following block statement is not erroneous, because the exception handler performs the same action regardless of the kind of exception that is propagated out of the parallelizable sequence of statements:

```
begin
    pragma Parallelizable_Sequence;
    A(I) := B(I)*C(I);
    A(J) := B(J)*C(J);
exception
    when others => Put_Line("Error");
end;
```

Additional examples of the restriction on shared mutable variables in a parallelizable sequence of statements are given in Section 3.5.

3.3 The Parallelizable For-Loop Statement Pragma

3.3.1 Syntax

A Parallelizable_Loop pragma is applied to a for-loop statement to indicate that the iterations of the loop can be executed as parallel threads. The form of an ordinary Parallelizable_Loop pragma is as follows:

```
pragma Parallelizable_Loop;
for loop_parameter in [reverse] loop_range loop
    loop_body
end loop;
```

A Parallelizable_Loop pragma is allowed only immediately preceding a for-loop statement. The for-loop statement is referred to as a parallelizable for-loop statement. In Section 3.7, we describe optional arguments of the pragma that may improve run-time performance.

3.3.2 Summary of the Standard Sequential Semantics

The sequential semantics of a parallelizable for-loop statement is the same as the standard sequential semantics of a for-loop statement. Execution of a for-loop statement consists of evaluation of the loop range, followed by execution of the loop body repeatedly, once for each value of the loop range or until a transfer of control out of the loop takes place. The loop parameter is implicitly declared and is only visible within the loop body. In each iteration, the loop parameter is a different constant from the loop range and cannot be changed by the execution of the loop body. The iterations are executed in increasing order of loop parameter value, unless the word **reverse** is present, in which case the iterations are executed in decreasing order of loop parameter value. It is illegal for a goto statement to transfer control into a for-loop statement.

Transfer of control out of a for-loop statement can be caused by: (i) execution of an exit statement, return statement, or goto statement within the loop body, or (ii) the raising of an exception within the loop body. Because of the restrictions described in Section 3.3.3, a transfer of control out of a parallelizable for-loop statement can only be caused by the raising of an exception within the loop body.

3.3.3 Restrictions

In order for the parallel semantics to be equivalent to the standard sequential semantics, a parallelizable for-loop statement is subject to the standard restrictions on a for-loop statement, plus a small number of additional restrictions, described below. These additional restrictions are the same as or analogous to the additional restrictions on a parallelizable sequence of statements, described in Section 3.2.3.

Restrictions on Transfer of Control:

- It is illegal for an exit statement, return statement, or goto statement to transfer control out of a parallelizable for-loop statement. (I.1)

Restrictions on Shared Variables:

- It is erroneous for one iteration of a parallelizable for-loop statement to assign a mutable variable that is assigned or evaluated by another iteration of the same parallelizable for-loop statement, unless those actions are synchronized. (E.1)

- Similarly, it is erroneous for one iteration of a parallelizable for-loop statement to deallocate any dynamically-allocated variable that is assigned or evaluated by another iteration of the same parallelizable for-loop statement, unless those actions are synchronized. (E.2)

These two restrictions are described in more detail in Section 3.5.

Restrictions on Exception Handling:

- It is erroneous to make a choice based on the kind of exception that is propagated out of a parallelizable for-loop statement that terminates exceptionally. (E.3)

- It is erroneous to evaluate or assign certain variables after a parallelizable for-loop statement that terminates exceptionally. This restriction is described in more detail in Section 3.6. (E.4)

3.3.4 Equivalent Parallel Semantics

A parallelizable for-loop statement has the following parallel semantics:

- Initiation of the execution of a parallelizable for-loop statement consists of the evaluation of the loop range, followed by the initiation of the execution of each iteration as a separate parallel thread of control. The order in which the execution of the iterations is initiated is undefined.

- If execution of all the iterations terminates normally, execution of the parallelizable for-loop statement terminates normally.

- If execution of one of the iterations terminates exceptionally, execution of the other iterations is aborted and execution of the parallelizable for-loop statement terminates exceptionally. The kind of exception that is propagated is undefined.

If all of the restrictions are satisfied, execution of a parallelizable for-loop statement according to this parallel semantics is equivalent to execution according to the standard sequential semantics.

3.3.5 Examples

The following is a simple example of a parallelizable for-loop statement:

```
pragma Parallelizable_Loop;
for I in 1 .. N loop
    Data(I) := 0;
end loop;
```

The following is an example of a parallelizable sequence of statements nested within a parallelizable for-loop statement:

```
pragma Parallelizable_Loop;
for I in 1 .. N loop
    pragma Parallelizable_Sequence;
    A(I) := 0;
    B(I) := 0;
end loop;
```

The following is illegal (violates restriction I.1), because the exit statement transfers control out of the parallelizable for-loop statement:

```
Found := False;
pragma Parallelizable_Loop;
for I in 1 .. N loop
    if Data(I) = Target then
        Found := True;
        exit;
    end if;
end loop;
```

If Data is an array of mutable components, execution of the following parallelizable for-loop statement is erroneous (violates restriction E.1), because each component of Data is assigned by one iteration and evaluated by another iteration:

```
Data(1) := 0;
pragma Parallelizable_Loop;
for I in 2 .. N loop
    Data(I) := Data(I − 1);
end loop;
```

Execution of the following block statement is erroneous (violates restriction E.3), because the exception handler makes a choice based on the kind of exception that is propagated out of the parallelizable for-loop statement:

```
begin
    pragma Parallelizable_Loop;
    for I in 1 .. N loop
        A(I) := B(I)*C(I);
    end loop;
exception
    when Numeric_Error => Put_Line("Overflow");
end;
```

In contrast, execution of the following block statement is not erroneous, because the exception handler performs the same actions regardless of the kind of exception that is propagated out of the parallelizable for-loop statement:

```
begin
    pragma Parallelizable_Loop;
    for I in 1 .. N loop
        A(I) := B(I)*C(I);
    end loop;
exception
    when others => Put_Line("Overflow");
end;
```

Additional examples of the restriction on shared mutable variables in a parallelizable for-loop statement are given in Section 3.5.

3.4 The Single-Assignment Type Pragma

3.4.1 Syntax

A Single_Assignment pragma is applied to a type declaration to indicate that assignment and evaluation operations on variables of that type can be used as implicit communication and synchronization operations between parallel threads. The form of a Single_Assignment pragma is as follows:

```
type identifier is type_definition;
pragma Single_Assignment(identifier);
```

A Single_Assignment pragma is allowed only in a declarative region, and the identifier argument must denote a type declaration in the same declarative region. A Single_Assignment pragma will normally be placed immediately following the type declaration to which it applies. A type that is named by a Single_Assignment pragma is referred to as a single-assignment type, and a variable of a single-assignment type is referred to as a single-assignment variable. A type that is not a single-assignment type is referred to as a mutable type, and a variable of a mutable type is referred to as a mutable variable.

3.4.2 Summary of the Standard Sequential Semantics

The sequential semantics of operations on single-assignment types and variables is the same as the standard sequential semantics of operations on mutable types and variables. A type declaration declares a distinct named type, and elaboration of a variable declaration creates a variable object. The default initial value of a scalar variable is undefined and of an access variable is **null**. Evaluation of a variable returns the current value of the variable, and assignment to a variable replaces the current value of the variable with a new value.

Explicit type conversions are allowed between related types such as a derived type and its parent type.

3.4.3 Restrictions

In order for the parallel semantics to be equivalent to the standard sequential semantics, single-assignment types and variables are subject to the standard restrictions on mutable types and variables, plus a small number of additional restrictions, described below.

Fundamental Restrictions on Assignment and Evaluation:

- It is erroneous to evaluate a single-assignment variable that has not previously been assigned. (E.5)

- It is erroneous to assign a single-assignment variable that has previously been assigned. This is the most significant restriction on single-assignment variables. (E.6)

Other Related Restrictions:

- It is illegal for a single-assignment type to be a limited type (i.e., a type without an assignment operation). (I.3)

- It is illegal to assign an individual component of a variable of a composite single-assignment type (i.e., a single-assignment array or record type). (I.4)

- It is illegal for an actual parameter of **in out** or **out** mode to be an individual component of a variable of a composite single-assignment type. (I.5)

- It is illegal for an actual parameter of **in out** or **out** mode to be a type conversion between a single-assignment type and a mutable type. (I.6)

- A single-assignment derived type does not inherit subprograms from its parent type in which the type is a formal parameter of **in out** or **out** mode. (I.7)

3.4.4 Equivalent Parallel Semantics

Operations on single-assignment types and variables have the following parallel semantics:

- The default initial value of a single-assignment variable is a special "unassigned" value.

- Assignment and evaluation of a single-assignment variable are atomic operations (i.e., they are not interleaved when executed in parallel).

- Evaluation of an unassigned single-assignment variable causes the evaluating thread to suspend at that point.
- Assignment to an unassigned single-assignment variable awakens all threads that are suspended on the evaluation of that variable.
- The effect of assignment to a previously assigned single-assignment variable is undefined. One possibility is to raise an implementation-defined exception.
- A single-assignment variable or expression occurring as an actual parameter of **in** mode is passed by value. A single-assignment variable occurring as an actual parameter of **in out** or **out** mode is passed by reference.

If all of the restrictions are satisfied, execution of operations on single-assignment types and variables according to this parallel semantics is equivalent to execution of operations on mutable types and variables according to the standard sequential semantics.

3.4.5 Examples

Type Declarations

The following are examples of type declarations involving single-assignment types:

```
-- a single-assignment integer type derived from the standard integer type
type Single_Integer is new Integer;
pragma Single_Assignment(Single_Integer);

-- an array type with single-assignment components
type Array_Of_Single is array (1 .. 10) of Single_Integer;

-- a single-assignment array type
type Single_Array is array (1 .. 10) of Integer;
pragma Single_Assignment(Single_Array);

-- a record type with single-assignment components
type Record_Of_Single is
    record
        X, Y, Z : Single_Integer;
    end record;
```

```
-- a record type with mutable and single-assignment components
type Record_Of_Single is
    record
        X : Integer;
        Y : Single_Integer;
    end record;

-- a single-assignment record type
type Single_Vector is
    record
        X, Y, Z : Integer;
    end record;
pragma Single_Assignment(Single_Vector);

-- a linked list with single-assignment links
type Node;
type Single_Pointer is access Node;
pragma Single_Assignment(Single_Pointer);
type Node is
    record
        Item : Integer;
        Next : Single_Pointer;
    end record;
```

It is important to understand the distinction between a composite type with single-assignment components and a single-assignment composite type. Assignment to the individual components of a composite type with single-assignment components is permitted, whereas a composite single-assignment variable must be assigned as a whole.

Sequential Assignment and Evaluation

The following is a simple example of sequential assignment and evaluation of single-assignment variables:

```
declare
    X, Y : Single_Integer;
begin
    X := 1;
    Y := X;
end;
```

Execution of the following block statement is erroneous (violates restriction E.3), because X is evaluated before it is assigned:

```
declare
    X, Y : Single_Integer;
begin
    Y := X;
    X := 1;
end;
```

Execution of the following block statement is erroneous (violates restriction E.4), because X is assigned more than once:

```
declare
    X : Single_Integer;
begin
    X := 1;
    X := 2;
end;
```

Parallelizable Assignment and Evaluation

The following is a simple example of parallelizable assignment and evaluation of single-assignment variables:

```
declare
    X, Y : Single_Integer;
begin
    pragma Parallelizable_Sequence;
    X := 1;
    Y := X;
end;
```

Execution of the following block statement is erroneous (violates restriction E.3), because X is evaluated before it is assigned (in the sequential interpretation):

```
declare
    X, Y : Single_Integer;
begin
    pragma Parallelizable_Sequence;
    Y := X;
    X := 1;
end;
```

3.5 Restrictions on Shared Variables

The two most significant restrictions on a parallelizable sequence of statements are the following:

1. It is erroneous for one statement of a parallelizable sequence of statements to assign a mutable variable that is assigned or evaluated by another statement of the same parallelizable sequence of statements, unless those actions are synchronized according to the parallel semantics. (E.1)

2. Similarly, it is erroneous for one statement of a parallelizable sequence of statements to deallocate any dynamically-allocated variable that is assigned or evaluated by another statement of the same parallelizable sequence of statements, unless those actions are synchronized according to the parallel semantics. (E.2)

Analogous restrictions apply to the iterations of a parallelizable for-loop statement. Action A_1 in thread T_1 is synchronized with action A_2 in thread T_2 if T_1 assigns to a single-assignment variable S after executing A_1 and T_2 evaluates S before executing A_2. For example, execution of the following parallelizable sequences of statements is erroneous (violates restriction E.1) because of unsynchronized operations on the mutable variable X:

```
declare
    X : Integer;
begin
    pragma Parallelizable_Sequence;
    X := 1;
    X := 2;
end;
```

```
declare
    X, Y : Integer;
begin
    pragma Parallelizable_Sequence;
    X := 1;
    Y := X;
end;
```

Execution of the following parallelizable sequence of statements is not erroneous, because synchronization on S ensures that the assignment to X by the first parallel thread occurs before the evaluation of X by the second parallel thread.

```
declare
    X, Y : Integer;
    S, T : Single_Integer;
begin
    pragma Parallelizable_Sequence;
    begin
        X := 1;
        S := 0;
    end;
    begin
        T := S;
        Y := X;
    end;
end;
```

In contrast, execution of the following parallelizable sequence of statements is erroneous (violates restriction E.1), because the operations on S do not synchronize the assignment and evaluation operations on X:

```
declare
    X, Y : Integer;
    S, T : Single_Integer;
begin
    pragma Parallelizable_Sequence;
    begin
        X := 1;
        S := 0;
    end;
    begin
        Y := X;
        T := S;
    end;
end;
```

In general, showing that these restrictions are satisfied requires reasoning about synchronization between parallel threads of control. However, in practice most instances of parallelizable sequences of statements (and parallelizable for-loop statements) satisfy the following two stronger restrictions defined in terms of the sequential semantics of the programming model:

1. It is erroneous for one statement of a parallelizable sequence of statements to assign a mutable variable that is assigned or evaluated by another statement of the same parallelizable sequence of statements.

2. Similarly, it is erroneous for one statement of a parallelizable sequence of statements to deallocate any dynamically-allocated variable that is assigned or evaluated by another statement of the same parallelizable sequence of statements.

Analogous restrictions apply to the iterations of a parallelizable for-loop statement. Therefore, in most cases the general restrictions on the shared variables of a parallelizable sequence of statements or a parallelizable for-loop statement can be shown to be satisfied by using sequential reasoning to show that the stronger restrictions are satisfied.

3.6 Restrictions on Exceptions

After a parallelizable sequence of statements terminates exceptionally, the following two restrictions apply:

1. It is erroneous to evaluate any mutable variable that could have been assigned by the execution of the parallelizable sequence of statements according to the parallel semantics, until that variable has subsequently been assigned. (E.3)

2. Similarly, it is erroneous to evaluate or assign any single-assignment variable that could have been assigned by the execution of the parallelizable sequence of statements according to the parallel semantics. (E.4)

Analogous restrictions apply after a parallelizable for-loop statement terminates exceptionally. In general, showing that these restrictions are satisfied requires reasoning about the parallel threads of control to determine which variables could be assigned values by execution according to the parallel semantics. However, sequential reasoning can be used to determine a superset of the variables that could been assigned values by parallel execution. In practice, most instances of recovery from an exception do not erroneously evaluate or assign any variable in this superset. Therefore, in most cases the general restrictions can be shown to be satisfied using sequential reasoning.

3.7 Parallelizable For-Loop Statement Arguments

In order to provide control over granularity and load balancing, a Parallelizable_Loop pragma has two optional arguments that are used to indicate that a parallelizable for-loop statement should be executed according to a modified parallel semantics. The modified parallel semantics is equivalent to the default parallel semantics, but the run-time performance may be different. The complete form of a Parallelizable_Loop pragma is as follows:

pragma Parallelizable_Loop
    [([Num_Threads =>] *integer*_expression, [Pattern =>] *pattern*_identifier)];

The arguments indicate that the iterations of the parallelizable for-loop statement should be executed as a specified number of parallel threads, instead of the default of a separate parallel thread for each iteration. The Num_Threads argument indicates the number of parallel threads, and the pattern argument indicates the method by which iterations are assigned to parallel threads. The pattern identifier can be Block, Cyclic, or On_Demand. An example of the assignment of iterations to threads for each of the different patterns is shown in Table 3.1.

The parallel semantics associated with the Block and Cyclic patterns can be defined by transformation of a parallelizable for-loop statements with arguments into an equivalent block statement enclosing a parallelizable for-loop statement without arguments. Therefore,

	Thread A	Thread B	Thread C	Thread D
Block	0, 1, 2	3, 4	5, 6, 7	8, 9
reverse/Block	9, 8	7, 6, 5	4, 3	2, 1, 0
Cyclic	0, 4, 8	1, 5, 9	2, 6	3, 7
reverse/Cyclic	9, 5, 1	8, 4, 0	7, 3	6, 2
On_Demand*	0, 6	1, 4, 8	2	3, 5, 7, 9
reverse/On_Demand*	9, 1, 0	8, 3, 2	7, 5, 4	6

*One possible assignment of iterations is given for On_Demand patterns.

Table 3.1: Example of the assignment of iterations in the loop range 0 .. 9 to four threads for a parallelizable for-loop statement with and without **reverse** and with different patterns.

these patterns are really a convenient syntactic shorthand, rather than a distinct feature of our parallel programming model. The parallel semantics associated with the On_Demand pattern introduces nondeterministic assignment of iterations to threads, but the iterations are always initiated in sequential order.

Block Pattern

The Block pattern indicates that iterations should be assigned to parallel threads in contiguous blocks.

```
pragma Parallelizable_Loop(Num_Threads => N, Pattern => Block);
for I in [reverse] First .. Last loop
    Loop_Body(I);
end loop;
```

A parallelizable for-loop statement without **reverse** and with the Block pattern is defined by transformation to the following:

```
declare
    Num_Iterations : constant Integer := Max(0, Pos(Last) − Pos(First) + 1);
    Num_Threads    : constant Integer := Min(Max(1, N), Num_Iterations);
begin
    pragma Parallelizable_Loop;
    for T in 0 .. Num_Threads − 1 loop
        for K in Integer((Float(T)/Float(Num_Threads))*Float(Num_Iterations)) ..
                Integer((Float(T + 1)/Float(Num_Threads))*Float(Num_Iterations)) − 1 loop
            Loop_Body(Val(Pos(First) + K));
        end loop;
    end loop;
end;
```

A parallelizable for-loop statement with **reverse** and with the Block pattern is defined by the same transformation except that the T and K loops are **reverse** loops.

Cyclic Pattern

The Cyclic pattern indicates that iterations should be assigned to parallel threads in a round-robin manner.

```
pragma Parallelizable_Loop(Num_Threads => N, Pattern => Cyclic);
for I in [reverse] First .. Last loop
    Loop_Body(I);
end loop;
```

A parallelizable for-loop statement without **reverse** and with the Cyclic pattern is defined by transformation to the following:

```
declare
    Num_Iterations : constant Integer := Max(0, Pos(Last) − Pos(First) + 1);
    Num_Threads    : constant Integer := Min(Max(1, N), Num_Iterations);
begin
    pragma Parallelizable_Loop;
    for T in 0 .. Num_Threads − 1 loop
        for K in 0 .. (Num_Iterations + Num_Threads − T − 1)/Num_Threads − 1 loop
            Loop_Body(Val(Pos(First) + T + K*Num_Threads));
        end loop;
    end loop;
end;
```

A parallelizable for-loop statement with **reverse** and with the Cyclic pattern is defined by the same transformation except that the T and K loops are **reverse** loops.

On_Demand Pattern

The On_Demand pattern indicates that an idle parallel thread should be assigned the next unexecuted iteration.

```
pragma Parallelizable_Loop(Num_Threads => N, Pattern => On_Demand);
for I in [reverse] First .. Last loop
    Loop_Body(I);
end loop;
```

A parallelizable for-loop statement without **reverse** and with the On_Demand pattern is defined by transformation to the following:

```
declare
    Num_Iterations : constant Integer := Max(0, Pos(Last) − Pos(First) + 1);
    Num_Threads    : constant Integer := Min(Max(1, N), Num_Iterations);
    protected Loop_Parameters is
        procedure Next (K : out Integer; Finished : out Boolean);
    private
        Count : Integer := 0;
    end Loop_Parameters;
    protected body Loop_Parameters is
        procedure Next (K : out Integer; Finished : out Boolean) is
        begin
            if Count = Num_Iterations then
                Finished := True;
            else
                K := Count; Count := Count + 1; Finished := False;
            end if;
        end Next;
    end Loop_Parameters;
begin
    pragma Parallelizable_Loop;
    for T in 0 .. Num_Threads − 1 loop
        declare
            Finished : Boolean;
            K        : Integer;
        begin
            Loop_Parameters.Next(K, Finished);
            while not Finished loop
                Loop_Body(Val(Pos(First) + K));
                Loop_Parameters.Next(K, Finished);
            end loop;
        end;
    end loop;
end;
```

The Loop_Parameters protected object is a construct from outside of our model (from Ada 95 tasking) that enforces mutual exclusion on calls to the Next procedure. A parallelizable for-loop statement with **reverse** and with the On_Demand pattern is defined by the same transformation except that Pos(First) + K is replaced by Pos(Last) − K.

3.8 Compilation and Error Handling Options

A compiler for our parallel programming notation should provide options to compile a program according to either the standard sequential semantics or the equivalent parallel semantics. In the absence of erroneous execution situations, sequential and parallel execution will produce the same results. Therefore, most program development, testing, and debugging can be performed on the sequential version of the program. The linker must check that all of the compilation units of a program have been compiled according to the same semantics.

Although compilation according to the sequential semantics can be achieved using a standard compiler that ignores unknown pragmas, a compiler that understands the pragmas can provide additional support for detecting erroneous execution situations:

- With compilation according to the sequential semantics, evaluation of an unassigned single-assignment variable can be detected at run time.
- With compilation according to either the sequential or the parallel semantics, multiple assignment to a single-assignment variable can be detected at run time.
- With compilation according to either the sequential or the parallel semantics, a parallelizable sequence of statements or parallelizable for-loop statement that terminates exceptionally can propagate a unique implementation-defined exception, and a choice based on this exception can be detected.

Detection of any of these errors should terminate program execution in a manner that provides as much debugging information as possible. As discussed in Section 3.9, for reasons of efficiency the programmer may sometimes choose to suppress detection of these erroneous execution situations.

The only erroneous execution situations that cannot be detected at low cost are the errors relating to parallel operations on shared variables and operations on variables after exceptions described in Section 3.5 and Section 3.6. These errors are very difficult to prohibit and expensive to detect in any notation that allows nested declaration scopes.

3.9 Related Pragmas

In addition to adding three new pragmas, we also extend two standard Ada pragmas for suppressing error checking and specifying thread priorities.

3.9.1 Suppressing Error Checking

The standard Suppress pragma is used to indicate that the compiler can omit certain specified run-time checks, e.g., checks on array indexing and division by zero. We extend the pragma to allow suppression of the run-time checks relating to our programming model. The form of these Suppress pragmas is as follows:

pragma Suppress(Single_Assignment_Check);
pragma Suppress(Parallel_Exception_Check);

If Single_Assignment_Check is suppressed, the checks on evaluation of unassigned single-assignment variables and multiple assignment to single-assignment variables should be omitted. If Parallel_Exception_Check is suppressed, the propagation of exceptions out of parallelizable sequences of statements and parallelizable for-loop statements should be omitted.

3.9.2 Specifying Thread Priorities

The standard Priority pragma can appear within a task specification to assign a scheduling priority to the execution of the task. The form of a Priority pragma is as follows:

pragma Priority(*integer*_expression);

We extend the Priority pragma to apply to the statements of parallelizable sequences of statements and the iterations of parallelizable for-loop statements. A Priority pragma can appear immediately before a statement of a parallelizable sequence of statements to assign a priority to the execution of the statement. A Priority pragma can appear immediately before the sequence of statements enclosed by a parallelizable for-loop statement to assign priorities to the execution of the iterations. Since the expression in the Priority pragma can involve the loop parameter value, different iterations can be assigned different priorities.

Assigning priorities to parallel threads of control does not affect the semantics of our programming model, but can often be used to improve load balancing and hence run-time performance. In particular, higher priorities can be assigned to threads with large workloads and to threads that produce data items on which many other threads depend.

Chapter 4

Equivalence of the Parallel and Sequential Semantics

In this chapter, we prove that the parallel and sequential semantics of our programming model are equivalent. More precisely, we prove that for any program that satisfies the restrictions on the pragmas, execution of the program according to the parallel semantics is equivalent to execution of the program according to the standard sequential semantics. This equivalence result forms the basis of our methodology for developing parallel programs using sequential reasoning, testing, and debugging.

4.1 Preliminaries

Before we prove the equivalence of the parallel and sequential semantics of our programming model, we first define what we mean by specifications, equivalence of programs, and parallel execution.

4.1.1 Specifications

The specification of a terminating program can be expressed as a precondition assertion and a postcondition assertion on the input and output variables of the program. A specification of this form can be written as:

$$\{\text{Precondition}\}\ \text{Program}\ \{\text{Postcondition}\}$$

The meaning of the specification is that if the precondition holds in the state in which execution of the program is initiated, execution of the program will terminate and the postcondition will hold in the state in which execution of the program terminates. The specification says nothing about the behavior of the program if the precondition does not hold in the initial state. Specifications of this form were introduced by Hoare [59] (based on earlier work by Floyd [39]), and are described in more detail by Gries [51].

4.1.2 Equivalence of Programs

In this thesis, where we say that two programs, P and Q, are equivalent, we mean that for any given specification, P satisfies the specification if and only if Q satisfies the specification.

$$\forall \text{ Pre, Post} : \{\text{Pre}\}\ \text{P}\ \{\text{Post}\} \Leftrightarrow \{\text{Pre}\}\ \text{Q}\ \{\text{Post}\}$$

In other words, we cannot distinguish between the two programs on the basis of their interaction with their input and output variables.

4.1.3 Parallel Execution

We have informally described parallel execution of a group of statements to be the individual statements executed as separate parallel threads of control. We more precisely define parallel execution of a group of statements to be equivalent to an arbitrary interleaving of the atomic actions of the individual statements. For our model, the interleaving of the actions does not need to be fair and the definition of an atomic action is arbitrary, except that evaluations and assignments of single-assignment variables must be atomic actions.

Parallel execution of a group of statements may consist of truly concurrent execution on separate processors, interleaved execution on a single processor, sequential execution on a single processor, or some combination of these alternatives. The key requirement is that the result of parallel execution is indistinguishable from the result of interleaved execution. Therefore, we can reason about parallel execution by reasoning about interleaved execution. In particular, we can prove that parallel execution of a group of statements satisfies a given specification, by proving that all interleaved executions satisfy the specification.

4.2 Equivalence Theorem

The equivalence theorem that we prove in this chapter is as follows:

$$\forall \text{ P, Pre, Post} : \{\text{Pre}\}\ \text{P}_\text{P}\ \{\text{Post}\} \Leftrightarrow \{\text{Pre}\}\ \text{P}_\text{S}\ \{\text{Post}\}$$

where:

P is a program that satisfies the restrictions on the pragmas,

Pre and Post are assertions on the input and output variables of P,

P_P is execution of P according to the parallel semantics, and

P_S is execution of P according to the standard sequential semantics.

In other words, for any program that satisfies the restrictions on the pragmas, execution of the program according to the parallel semantics is equivalent to execution of the program according to the standard sequential semantics.

4.3 Equivalence Proof

The equivalence theorem is a consequence of Lemma 1 and Lemma 2.

Lemma 1

$$\forall \text{ P, Pre, Post} : \{\text{Pre}\}\ P_P\ \{\text{Post}\} \Rightarrow \{\text{Pre}\}\ P_S\ \{\text{Post}\}$$

Proof Outline: Any sequential execution of P is one possible parallel execution of P. Therefore, any specification that is satisfied by parallel execution of S is also satisfied by sequential execution of S.

Lemma 2

$$\forall \text{ P, Pre, Post} : \{\text{Pre}\}\ P_S\ \{\text{Post}\} \Rightarrow \{\text{Pre}\}\ P_P\ \{\text{Post}\}$$

Proof Outline: Any parallel execution of P is equivalent to some sequential execution of P, by a succession of swaps of pairs of out-of-order actions in the parallel interleaving. Therefore, any specification that is satisfied by sequential execution of S is also satisfied by parallel execution of S.

4.4 Lemma 1 Proof

Lemma 1 is a consequence of Lemma 1.1 and Lemma 1.2.

Lemma 1.1

$$\forall \text{ S, Pre, Post} : \{\text{Pre}\}\ S_P\ \{\text{Post}\} \Rightarrow \{\text{Pre}\}\ S_S\ \{\text{Post}\}$$

where S is a parallelizable sequence of statements that satisfies the restrictions on the pragmas, S_P is execution of S according to the parallel semantics, and S_S is execution of S according to the standard sequential semantics.

Lemma 1.1 Proof

Sequential execution of S is one possible parallel interleaving of S. Therefore, any specification that is satisfied by parallel execution of S is also satisfied by sequential execution of S.

Lemma 1.2

$$\forall \text{ L, Pre, Post} : \{\text{Pre}\}\ L_P\ \{\text{Post}\} \Rightarrow \{\text{Pre}\}\ L_S\ \{\text{Post}\}$$

where L is a parallelizable for-loop statement that satisfies the restrictions on the pragmas, L_P is execution of L according to the parallel semantics, and L_S is execution of L according to the standard sequential semantics.

Lemma 1.2 Proof

Sequential execution of L is one possible parallel interleaving of L. Therefore, any specification that is satisfied by parallel execution of L is also satisfied by sequential execution of L.

4.5 Lemma 2 Proof

Lemma 2 is a consequence of Lemma 2.5 and Lemma 2.6. The proof is related to the diamond property and the Church-Rosser theorem [20][26] which is the theoretical basis of parallel functional programming systems.

Lemma 2.1

Consider any parallelizable sequence of two statements that satisfies the restrictions on the pragmas:

```
pragma Parallelizable_Sequence;
A; B;
```

Consider any parallel interleaving of the actions of the two statements, for example:

$$B_1; A_1; A_2; B_2; \ldots; A_{n-1}; B_{m-1}; B_m; A_n;$$

Consider any adjacent pair of actions that are out of sequential order in the interleaving:

$$\ldots B_i; A_j; \ldots$$

The pair of actions is equivalent to the pair of actions swapped into sequential order:

$$\ldots A_j; B_i; \ldots$$

Lemma 2.1 Proof

Each of the actions can be categorized as either:

1. $Read\ X;$

   An evaluation operation on a single-assignment variable X.

2. $Write\ X;$

   An assignment operation on a single-assignment variable X.

3. $Mutable;$

   Any operation on mutable variables.

Consider all the possibilities for an adjacent pair of actions that are out of sequential order in the interleaving, where X and Y are different single-assignment variables:

1. $Read\ X; Read\ X;$ or
   $Read/Write\ X; Read/Write\ Y;$

   The pair of actions is equivalent to the swapped pair of actions.

2. $Read\ X; Write\ X;$

   The program violates restriction E.6 regarding multiple assignment to single-assignment variables, since X must have previously been assigned.

3. $Write\ X; Read\ X;$

   If X has not previously been assigned, the program violates restriction E.5 regarding evaluation of unassigned single-assignment variables. If X has previously been assigned, the program violates restriction E.6 regarding multiple assignment to single-assignment variables.

4. $Write\ X; Write\ X;$

   The program violates restriction E.6 regarding multiple assignment to single-assignment variables.

5. $Read/Write\ X; Mutable;$ or
   $Mutable; Read/Write\ X;$ or
   $Mutable; Mutable;$

   If restrictions E.1 and E.2 regarding operations on shared variables are not violated, the pair of actions is equivalent to the swapped pair of actions.

Lemma 2.2

Consider any parallelizable sequence of two statements that satisfies the restrictions on the pragmas:

```
pragma Parallelizable_Sequence;
A; B;
```

Consider any parallel interleaving of the actions of the two statements, for example:

$$B_1; A_1; A_2; B_2; \ldots; A_{n-1}; B_{m-1}; B_m; A_n;$$

Any parallel interleaving of the actions of the two statements is equivalent to the sequential ordering of the actions:

$$A_1; A_2; \ldots; A_{n-1}; A_n; B_1; B_2; \ldots; B_{m-1}; B_m;$$

Lemma 2.2 Proof

Any parallel interleaving of the actions of the two statements is equivalent to the sequential ordering of the actions, by a succession of swaps of adjacent actions that are out of sequential order in the interleaving. By Lemma 2.1, each swap maintains the invariant that the interleaving is equivalent to the initial parallel interleaving. When no further swaps are possible, the interleaving is the sequential ordering of the actions.

To see that the number of swaps is finite, let M be the total number of pairs of actions (not restricted to adjacent pairs of actions) that are out of sequential order in the interleaving. Since we consider only terminating programs, the length of the interleaving is finite, and hence M is finite. Each swap produces an equivalent interleaving in which M is decreased. Since M is bounded below by zero, the number of swaps is finite.

Lemma 2.3

Consider any parallelizable sequence of N statements that satisfies the restrictions on the pragmas:

pragma Parallelizable_Sequence;
A^1; A^2; ...; A^{N-1}; A^N;

Any parallel interleaving of the actions of the N statements is equivalent to the sequential ordering of the actions.

Lemma 2.3 Proof

The proof is by induction on N.

Base Case: Lemma degenerately holds for $N = 1$.

Inductive Hypothesis: Lemma holds for all $N < K$, for some $K > 1$.

Induction Step: $N = K$. By the inductive hypothesis, any parallel interleaving of the actions of A^1; ...; A^{K-1}; is equivalent to the sequential ordering of the actions. Therefore, by Lemma 2.2 with A equal to A^1; ...; A^{K-1}; and B equal to A^K, any parallel interleaving of the actions of A^1; ...; A^K; is equivalent to the sequential ordering of the actions.

Lemma 2.4

For any parallelizable sequence of statements S that satisfies the restrictions on the pragmas, if some parallel execution of S terminates exceptionally, there exists some sequential execution of S that terminates exceptionally.

Lemma 2.4 Proof

By Lemma 2.3, the actions of the parallel execution of S are equivalent to the sequential ordering of those actions. If statement A^k terminates exceptionally in parallel execution, there exists a sequential execution in which statement A^i with $i \leq k$ terminates exceptionally. The actions of the statements $A^1 \ldots A^i$ in the sequential reordering of the parallel execution of S form the prefixes of the actions of the statements $A^1 \ldots A^i$ in the sequential execution of S.

Lemma 2.5

$$\forall \text{ S, Pre, Post} : \{\text{Pre}\}\ \text{S}_\text{S}\ \{\text{Post}\} \Rightarrow \{\text{Pre}\}\ \text{S}_\text{P}\ \{\text{Post}\}$$

where S is a parallelizable sequence of statements that satisfies the restrictions on the pragmas, S_S is execution of S according to the standard sequential semantics, and S_S is execution of S according to the parallel semantics.

Lemma 2.5 Proof

Any parallel execution of S either terminates normally or terminates exceptionally.

Case 1: Parallel execution of S terminates normally.

By Lemma 2.3, a parallel execution of S that terminates normally is equivalent to some sequential execution of S.

Case 2: Parallel execution of S terminates exceptionally.

By Lemma 2.4, if the parallel execution of S terminates exceptionally, there exists some sequential execution of S that terminates exceptionally. Any postcondition that depends on the kind of exception raised or the state of the variables modified by the execution of S violates restriction E.3 or E.4. Therefore, a parallel execution of S that terminates exceptionally is equivalent to some sequential execution of S.

In both cases, any parallel execution of S is equivalent to some sequential execution of S. Therefore, any specification that is satisfied by sequential execution of S is also satisfied by parallel execution of S.

Lemma 2.6

$$\forall \text{ L, Pre, Post} : \{\text{Pre}\}\ \text{L}_\text{S}\ \{\text{Post}\} \Rightarrow \{\text{Pre}\}\ \text{L}_\text{P}\ \{\text{Post}\}$$

where L is a parallelizable for-loop statement that satisfies the restrictions on the pragmas, L_S is execution of L according to the standard sequential semantics, and L_S is execution of L according to the parallel semantics.

Lemma 2.6 Proof

Case 1: Parallelizable For-Loop Statements without Arguments.

A parallelizable for-loop statement without arguments is equivalent to a parallelizable sequence of statements in which the number of statements is computed immediately prior to execution of the sequence of statements. Therefore, Lemma 2.6 for parallelizable for-loop statements without arguments follows from Lemma 2.5.

Case 2: Parallelizable For-Loop Statements with Arguments.

The arguments of a parallelizable for-loop statement simply restrict the set of parallel interleavings. Therefore, Lemma 2.6 for parallelizable for-loop statements with arguments follows from Lemma 2.6 for parallelizable for-loop statements without arguments.

Chapter 5

Experimental Methods and Performance Issues

In this chapter, we describe the computer system, compiler, and methods that we use in developing and measuring the performance of experimental programs for this thesis. We also discuss the issues that affect the performance of parallel programs developed in this context. The computer system is a 36-processor shared-memory multiprocessor. On this platform, we implement our parallel programming model by transformation into tasking constructs, which are compiled into calls to a thread library. In Chapter 6 and Chapter 7, we present a collection of experimental programs and performance measurements.

5.1 Computer System

Our experiments are performed on from 1 to 32 processors of a 36-processor SGI Challenge computer system running the Irix 6.1 operating system. Details of this system are given in Table 5.1. The SGI Challenge is a symmetric multiprocessor. Each processor has its own local first-level and second-level cache memories and access to a shared main memory. Main memory is interleaved to allow multiple concurrent memory accesses. The hardware is entirely responsible for the transfer of data between main memory and cache memory, and for maintaining cache coherence between the processors. The SGI Challenge architecture is described by Galles and Williams [47] and in SGI technical documentation [108].

Our choice of a shared-memory architecture as the primary hardware platform for our performance experiments is based in part on the convenience of the hardware support for our shared-memory programming model, and also on our belief in the importance of this

Name	cydrome.mti.sgi.com
Kind	SGI Challenge
Operating system	Irix 6.1
Processors	36 × 100 MHz MIPS R4400
L1 cache	16 Kbytes data, 16 Kbytes instruction
L1 cache line size	16 bytes
L2 cache	1 Mbyte combined data and instruction
L2 cache line size	128 bytes
Main memory	768 Mbytes, 2-way interleaved
Cache policy	write back, snoopy cache coherence

Table 5.1: Computer system details.

class of parallel computer systems. In the near future, we expect to see the proliferation of small-scale to moderate-scale symmetric multiprocessor servers, workstations, and personal computers. This will be mostly driven by the ease of exploiting concurrency between multiple programs and between multiple users. As the number of installed multiprocessor systems increases, so will the market for programs that can transparently execute across multiple processors on these systems. Examples of applications that would benefit from multiprocessor support include commodity software programs such as spreadsheets, computer-aided design tools, interactive symbolic computation systems, and simulation packages.

5.2 Compilation

We implement the parallel semantics of our programming model by manual transformation into equivalent standard Ada 95 tasking constructs, as follows:

1. A parallelizable sequence of statements is transformed into a block statement containing a sequence of tasks, with one task for each statement.

2. A parallelizable for-loop statement is transformed into a block statement containing an array of tasks, with one task for each parallel thread.

3. A single-assignment variable is transformed into a variable with an associated protected object that synchronizes assignment and evaluation operations.

Details of these transformations are presented in Appendix A. They are also discussed in earlier work by the author of this thesis [126][129].

The programs are compiled with the SGI-Irix release of the GNAT (GNU-NYU Ada Translator) compiler [105]. GNAT is an Ada 95 front-end and run-time system for the GCC

(GNU C Compiler) family of compilers [116]. The efficiency of sequential code produced by GNAT is comparable to that produced by GCC for C/C++ programs (except that dynamically-sized arrays are implemented inefficiently in the current release). This to be expected, since Ada 95 is a conventional imperative language with sequential constructs and capabilities similar to C/C++, and the GNAT compilation system uses the standard GCC code generator and optimizer as its back end. For our performance experiments, all programs are compiled with the `-O2` optimization option.

The current release of GNAT implements dynamically-sized arrays inefficiently. The execution time of quicksort of a one dimensional array of integers is approximately three times that of an equivalent C program, because of Ada bounds checking and the inefficient implementation of arrays. The execution times of matrix multiplication and LU factorization of two dimensional arrays of floating point numbers are approximately six times that of equivalent C programs, because of Ada bounds checking and the inefficient implementation of arrays. However, this inefficiency does not affect the validity of our results, since sequential and parallel programs are slowed down equally.

GNAT compiles Ada tasking constructs into calls to the SGI-Irix implementation of the Pthreads (POSIX threads) library [97], as described by Giering, Muller, and Baker [49]. At run time, threads are dynamically scheduled across a user-specified number of sprocs (system processes), and sprocs are dynamically scheduled across processors. Sprocs share the pool of processors with the other processes running concurrently on the system. At different times, a thread may execute on different sprocs, and an sproc may execute on different processors, but the maximum number of processors executing a given program at any time is limited by the number of sprocs. By default, scheduling is nonpreemptive among threads of the same priority.

Implementation of our programming model on top of the GNAT implementation of Ada tasking provides us with a convenient means of testing programs and measuring their performance. However, this is not a particularly efficient method of implementation. We pay performance penalties for: (i) the cost of Ada tasking features that are not required by our threads, and (ii) the cost of the mapping from the Ada tasking model to the Pthreads model. The performance of our prototype implementation strategy could be improved upon significantly through the use of a more direct method of implementation. Nonetheless, this simple strategy is sufficient for us to demonstrate good speedups on medium-grained parallel programs.

5.3 Performance of Parallel Constructs

When attempting to determine an efficient level of granularity for a parallel program, it is useful to have a quantified understanding of the approximate run-time overheads of the fundamental parallel operations. In this section, we present the results of experiments that measure the execution time of our parallel programming constructs in this prototype implementation of our parallel programming model. It is not the purpose of this section to provide a comprehensive or precise benchmark of parallel or sequential performance.

5.3.1 Sequential Operations

As a basis for comparison, in the following table, we first present measurements of the execution time of some fundamental sequential operations:

Construct	Execution time (microseconds)
X := A(I); from L1 cache	0.04
X := A(I); from L2 cache	0.19
X := A(I); from main memory	1.19
X := Y*Z;	0.02
for I **in** 1 .. N **loop** ...	0.04N
P0;	0.21
P1(X);	0.25

X is a floating-point variable in the L1 cache, A is a statically-sized, zero-based array of floating-point components, I is an integer variable in a register, Y and Z are floating-point variables in registers, P0 is a procedure with no parameters and a null body, and P1 is a procedure with one **in out** mode parameter and a null body. All variables are global variables, and all operations are performed at the outermost level of the program, with index and range checking suppressed.

5.3.2 Parallelizable Sequences of Statements

In the following table, we present measurements of the execution time of a parallelizable sequence of null statements, executed according to the parallel semantics on a single processor, with and without parallel errors checked:

Number of statements	Execution time (milliseconds)	
	Checked	Unchecked
1	6.07	3.50
2	8.53	5.62
4	13.92	10.52
6	18.53	15.81
8	23.57	21.74
10	29.37	26.21

For small numbers of statements, the execution time of a parallelizable sequence of statements is approximately linearly dependent on the number of statements, as shown in Figure 5.1. The execution overhead of a parallel thread is many thousands of fundamental sequential operations. Therefore, very fine-grained parallelism is not efficient in this prototype implementation of our parallel programming model.

5.3.3 Parallelizable For-Loop Statements

In the following table, we present measurements of the execution time of a simple parallelizable for-loop statement with a null body, executed according to the parallel semantics on a single processor, with and without parallel errors checked:

Number of iterations	Execution time (milliseconds)	
	Checked	Unchecked
1	5.92	3.47
2	8.77	5.70
5	16.33	13.71
10	29.14	25.07
20	53.75	52.57
50	175.23	138.34
100	377.45	287.62
200	954.12	876.71
300	1571.12	1448.30
400	2477.09	2427.66
500	3342.36	3218.67

For small numbers of iterations, the execution time of a parallelizable sequence of statements is approximately linearly dependent on the number of iterations. However, for larger numbers of iterations, the execution time increases more than linearly with increasing numbers of iterations, as shown in Figure 5.2. Therefore, only a moderate number of parallel threads can be supported efficiently in this prototype implementation of our parallel programming model.

In the following table, we present measurements of the execution time per iteration within one thread of a parallelizable for-loop statement with a null body and different pattern arguments:

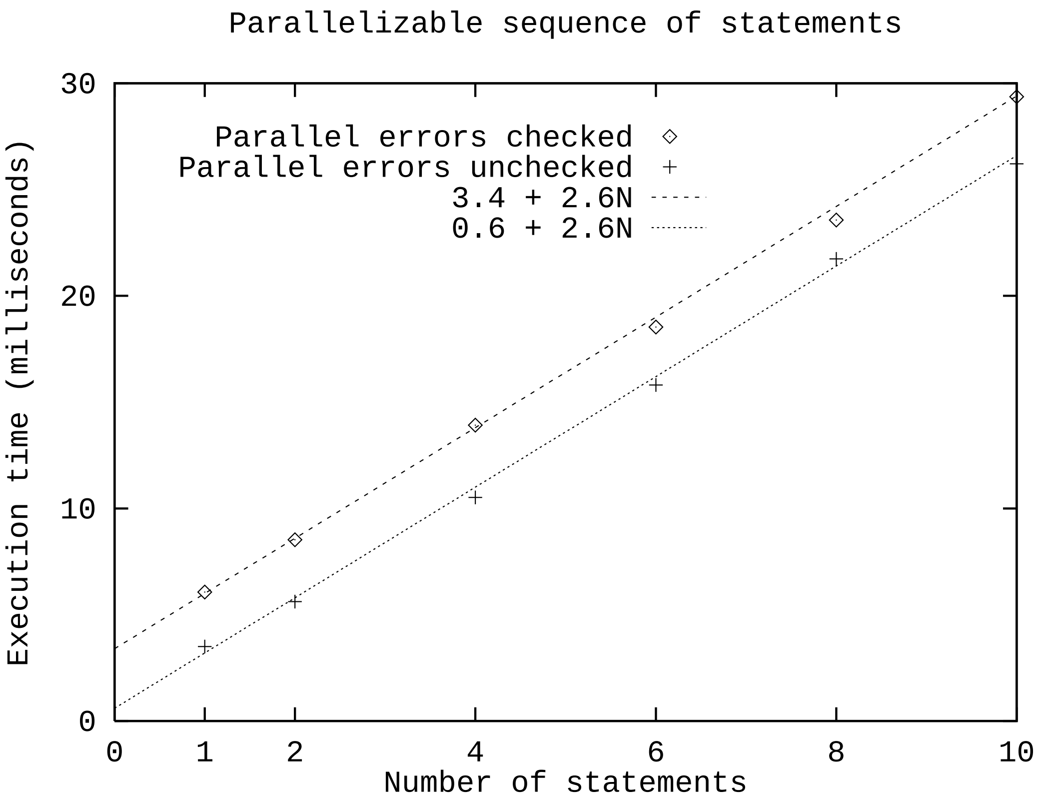


Figure 5.1: Execution time of a parallelizable sequence of statements on a single processor.

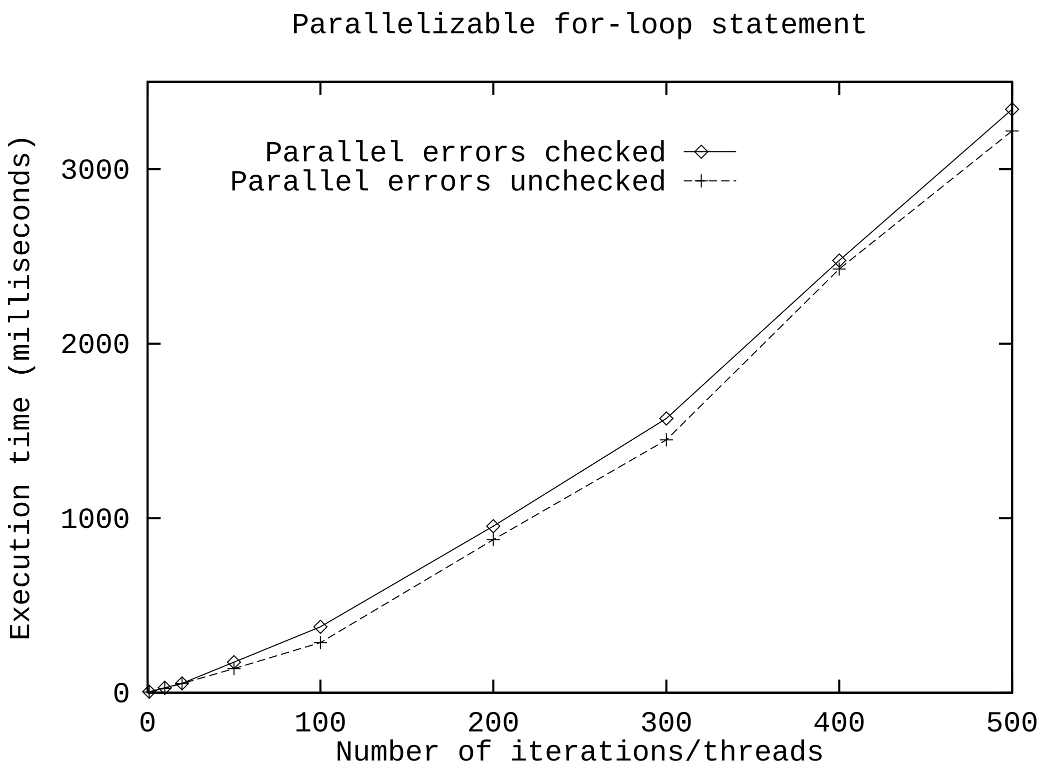


Figure 5.2: Execution time of a parallelizable for-loop statement on a single processor.

Pattern	Execution time per iteration (microseconds)
Block	0.04
Cyclic	0.04
On_Demand	15.00

The execution time per iteration is many hundred times greater for the On_Demand pattern than for the Block and Cyclic patterns, because of the synchronization operation required to obtain each loop parameter value. Therefore, a parallelizable for-loop statement that uses the On_Demand pattern with a large number of fine-grained iterations is not efficient in this prototype implementation of our parallel programming model.

5.3.4 Single-Assignment Variables

In the following table, we present measurements of the execution time of some fundamental operations on a single-assignment floating-point variable, executed on a single processor:

Operation	Execution time (microseconds)
Declaration	168
Assignment (checked)	74
Assignment (unchecked)	28
Evaluation	35

The measurements are for: (i) the implicit initialization and finalization operations associated with the declaration of a variable, (ii) assignment to a variable that has not previously been assigned a value (with checks), (iii) assignment to a variable that has not previously been assigned a value (with checks suppressed), and (iv) evaluation of a variable that has previously been assigned a value. The execution time of an operation on a single-assignment variable is tens to thousands of times greater than the execution time of an equivalent operation on a mutable variable. Therefore, very fine-grained use of single-assignment variables is not efficient in this prototype implementation of our parallel programming model.

5.4 Parallel Performance Issues

Development of an efficient parallel program in our parallel programming model requires consideration of several important issues that affect parallel performance. In this section, we summarize these issues in the context of execution on a shared-memory multiprocessor.

5.4.1 Granularity and Load Balancing

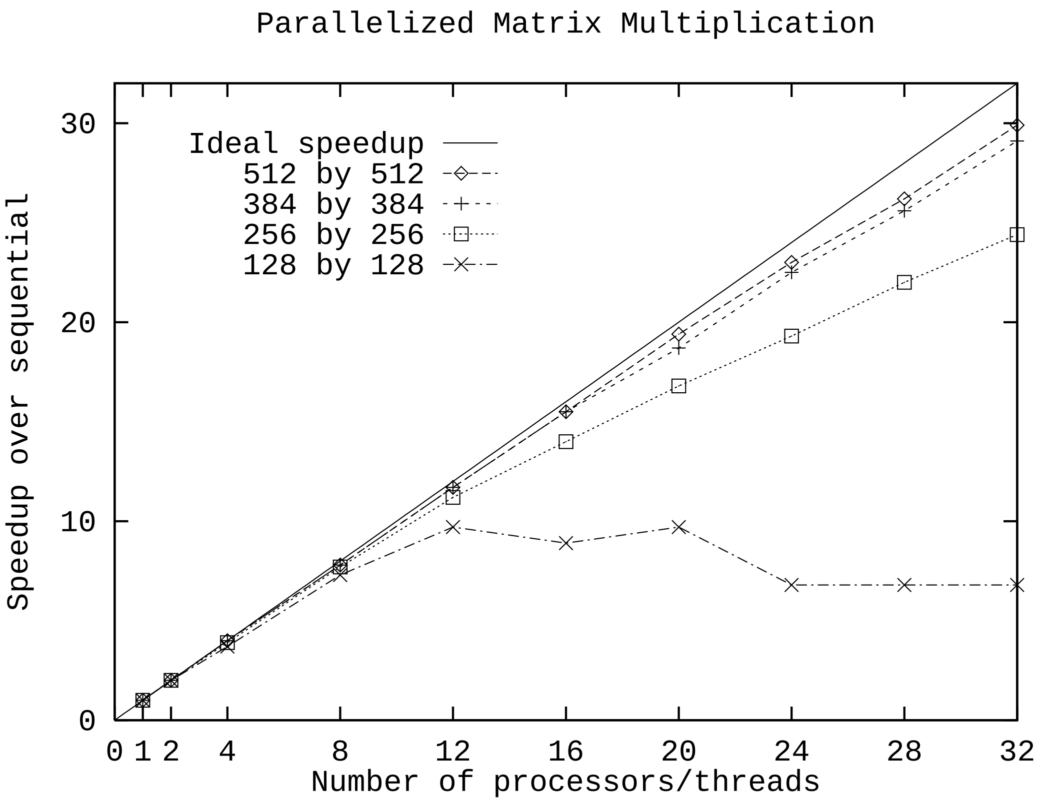
The granularity of a parallel program is related to the ratio of sequential computation to thread creation, termination, communication, and synchronization operations. A program is "fine-grained" if a relatively small amount of sequential computation is performed between thread operations and is "coarse-grained" if a relatively large amount of sequential computation is performed between thread operations. In our parallel programming model, the granularity of a program is determined by the number of parallel threads, the longevity of parallel threads, and the frequency with which threads access single-assignment variables.

Finding an efficient level of granularity for a parallel program involves a trade-off between processor load balancing and the execution overhead of parallelism. If a parallel program is too coarse-grained, there may be fewer executable threads than available processors for a large percentage of the computation. If a parallel program is too fine-grained, a large percentage of the total computation may be spent executing thread creation, termination, communication, and synchronization operations. Both cases limit the parallel speedup that can be achieved. An efficient balance can be found by program analysis or by experimentation.

Problem size is an important factor in determining an efficient level of granularity and in determining the maximum number of processors that can be used effectively. For example, the Multiply procedure in Program 5.1 multiplies two matrices using a straightforward matrix multiplication algorithm with the outer loop parallelized. In Figure 5.3, we compare the execution times and speedups of parallelized matrix multiplication for square matrices of varying sizes. In each case, the number of threads is equal to the number of processors, because this was found to give the best performance. For a given number of processors, speedup increases with matrix size, because the overhead of parallelism becomes less significant compared to the total computation. For the 128 by 128 case, speedup peaks then decreases with increasing numbers of processors, because the increasing overhead of parallelism begins to overwhelm the useful computation. In general, it is easier to obtain good speedups for larger problems than for smaller problems, particularly on a larger number of processors.

5.4.2 Locality and Caching

On any modern computer system (uniprocessor or multiprocessor), locality of data access is an extremely important factor in the performance of a program, due to the automatic caching of data by the memory system. For example, on our SGI Challenge target archi-



Number of processors	128 by 128		256 by 256		384 by 384		512 by 512	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
sequential	1.16	—	9.26	—	31.44	—	75.56	—
1	1.17	1.0	9.25	1.0	31.51	1.0	75.60	1.0
2	0.59	2.0	4.63	2.0	15.92	2.0	37.80	2.0
4	0.31	3.7	2.35	3.9	7.93	4.0	18.95	4.0
8	0.16	7.3	1.21	7.7	4.04	7.8	9.65	7.8
12	0.12	9.7	0.83	11.2	2.69	11.7	6.46	11.7
16	0.13	8.9	0.66	14.0	2.03	15.5	4.88	15.5
20	0.12	9.7	0.55	16.8	1.68	18.7	3.92	19.4
24	0.17	6.8	0.48	19.3	1.40	22.5	3.28	23.0
28	0.17	6.8	0.42	22.0	1.23	25.6	2.88	26.2
32	0.17	6.8	0.38	24.4	1.08	29.1	2.53	29.9

All times are in seconds.

Figure 5.3: Comparison of parallelized matrix multiplication for square matrices of varying sizes.

```
type Matrix is array (0 .. N − 1, 0 .. N − 1) of Float;

procedure Multiply (Num_Threads : in     Positive;
                    A, B        : in     Matrix;
                    Result      :    out Matrix  ) is
begin
   pragma Parallelizable_Loop(Num_Threads, Pattern => Block);
   for I in 0 .. N − 1 loop
      for J in 0 .. N − 1 loop
         declare
            Sum : Float := 0.0;
         begin
            for K in 0 .. N − 1 loop
               Sum := Sum + A(I, K)*B(K, J);
            end loop;
            Result(I, J) := Sum;
         end;
      end loop;
   end loop;
end Multiply;
```

Program 5.1: Parallelized matrix multiplication.

tecture, an access to main memory takes approximately 40 times as long as an access to first-level cache memory. Caching reduces the number of main memory accesses by transferring data between main memory and cache in "lines" (i.e., blocks of data) and by keeping recently (or frequently) accessed data in cache. The effectiveness of caching is determined by the degree of locality in the data access patterns of a program. For example, the performance of a program may vary significantly depending on whether the components of a matrix are accessed row by row or column by column.

On a shared-memory multiprocessor system, caching is particularly important, because it attempts to prevent the shared main memory from becoming a performance bottleneck. However, the performance of a parallel program may be affected by the overhead of maintaining coherent caches across processors. For example, if the caches of two processors contain the same data item and the first processor writes to the data item, the data item must be invalidated in the cache of the second processor. Performance is affected both by the cost of the invalidation operation and by the cost of any subsequent read of the data item by the second processor. Cache coherence mechanisms for shared-memory multiprocessors are surveyed by Lilja [83]. The key issues are summarized by Almasi and Gottlieb [5, section 10.3.2].

In our parallel programming model, it is erroneous for threads to share mutable data in the manner described above, between synchronization operations. Since synchronization operations are required to be relatively infrequent for reasons of efficiency, the cost of cache invalidation is insignificant in most practical programs. However, "false sharing" of data can result in situations where cache invalidation degrades parallel performance [130]. For example, consider Program 5.2, in which there is no sharing of variables between the two parallel threads. If the variables X and Y reside in different cache lines, the program can be

```
declare
    X, Y : Integer := 0;
begin
    pragma Parallelizable_Sequence;
    for I in 1 .. 1_000_000 loop
        X := X + 1;
    end loop;
    for J in 1 .. 1_000_000 loop
        Y := Y + 1;
    end loop;
end;
```

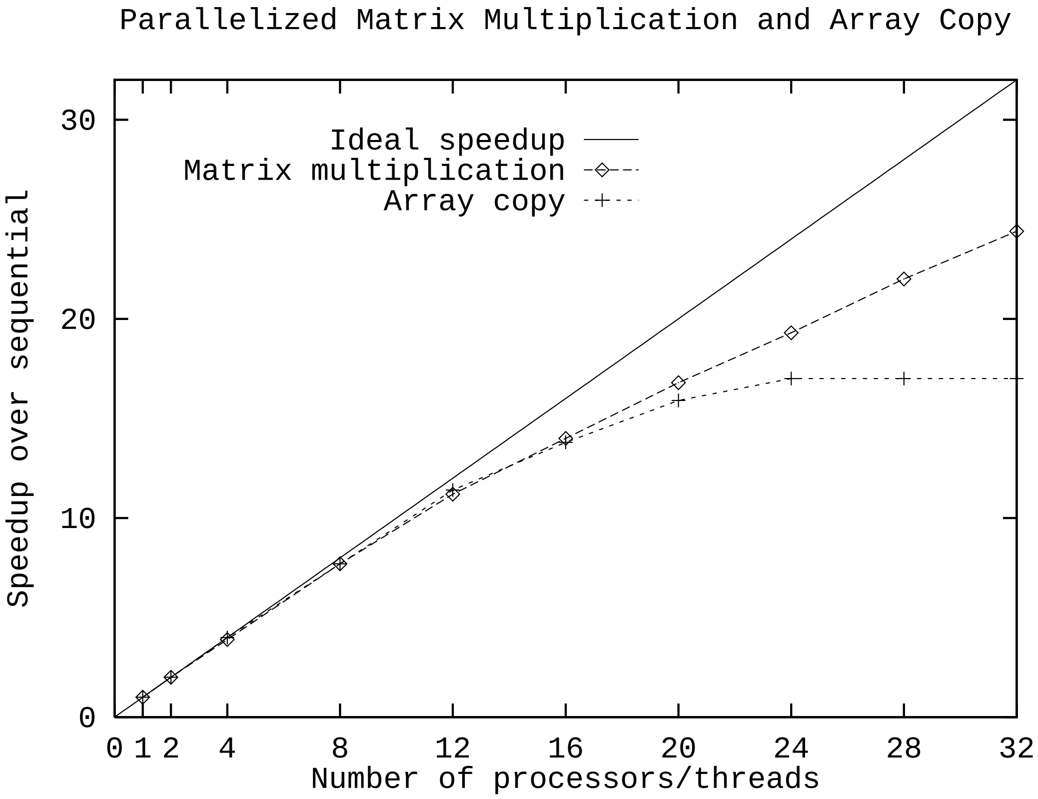
Program 5.2: An example of a program that could exhibit false sharing.

expected to yield almost perfect speedup (relative to sequential execution) when executed on two processors. If the variables X and Y reside in the same cache line, the program may actually slow down significantly (relative to sequential execution) when executed on two processors.

5.4.3 Memory Contention

Contention between processors for access to the shared main memory is a potential performance bottleneck that can limit parallel speedups [13][46]. The keys to whether or not memory contention is a significant problem are the memory bandwidth of the computer system and the ratio of local computation to main memory accesses in each parallel thread of the program. For example, the Copy procedure in Program 5.3 consists of a parallelized loop that copies the components of one array to another array. In Figure 5.4, we compare the execution times and speedups of the parallelized matrix multiplication given in Program 5.1 and parallelized array copying. The problem sizes are chosen so that the sequential execution times are approximately equal for the two programs.

The speedups for matrix multiplication continue to increase for up to 32 processors,



Number of processors	Matrix multiply		Array copy	
	Time	Speedup	Time	Speedup
sequential	9.26	—	10.36	—
1	9.25	1.0	10.38	1.0
2	4.63	2.0	5.20	2.0
4	2.35	3.9	2.62	4.0
8	1.21	7.7	1.35	7.7
12	0.83	11.2	0.91	11.4
16	0.66	14.0	0.75	13.8
20	0.55	16.8	0.65	15.9
24	0.48	19.3	0.61	17.0
28	0.42	22.0	0.61	17.0
32	0.38	24.4	0.61	17.0

All times are in seconds.

Figure 5.4: Comparison of parallelized matrix multiplication (256 by 256 components) and parallelized array copy (40 million components).

```
type Float_Array is array (0 .. N − 1) of Float;

procedure Copy (Num_Threads : in      Positive;
                To          :     out Float_Array;
                From        : in      Float_Array ) is
begin
   pragma Parallelizable_Loop(Num_Threads, Pattern => Block);
   for I in 0 .. N − 1 loop
      To(I) := From(I);
   end loop;
end Copy;
```

Program 5.3: Parallelized array copy.

whereas the speedups for array copying level off above 20 processors. The reason is that each thread of the matrix multiplication reads its operands into cache and performs significant amounts of local computation using the cached values. In contrast, each thread of the array copy reads or writes each component only once, thereby limiting the effectiveness of caching. Nonetheless, the cache line size and memory interleaving still result in good speedups, even for this memory-intensive program. Memory contention would be more significant in a program where the parallel threads accessed a large data set in a pattern without any locality.

5.4.4 Process and Data Mapping

On a distributed-memory multiprocessor, the mapping of data to local memories and the mapping of threads to processors are extremely important issues in parallel performance. On a symmetric shared-memory multiprocessor, these issues are controlled by the computer system rather than the programmer. All data resides in the shared memory, and threads are dynamically scheduled from a single pool. For moderate-scale multiprocessors, this strategy appears to be both convenient for the programmer and generally efficient.

5.5 Experimental Methods

We use the following methods in gathering performance measurements for this thesis:

- Experiments are performed on an otherwise unloaded machine.

- Unless otherwise stated, exception checking is not suppressed.

- Times are elapsed "wall-clock" times, measured using the standard Ada Clock function.
- Measurements are averaged over many trials, with high and low outliers discarded.
- Parallel speedups are relative to execution of a separately developed sequential program, not to the parallel program executed on a single processor.
- All experiments are repeated at least once to verify the consistency of the results.

The full text of the experimental parallel programs and the sequential programs that are used for performance comparison is presented in Appendix B.

Chapter 6

Experiments Using Parallelizable Sequences and For-Loops

In this chapter, we describe parallel programming experiments designed to investigate the use of parallelizable sequences of statements and parallelizable for-loop statements without additional synchronization. The experiments are performed in the context of the hardware and software platform described in Chapter 5. In Chapter 7, we describe parallel programming experiments designed to investigate the use of single-assignment types for additional synchronization.

6.1 Experimental Goals

The programming experiments that we describe in this chapter investigate parallel programming using parallelizable sequences of statements and parallelizable for-loop statements without additional synchronization. The goal of these experiments is to evaluate the following propositions with respect to this programming model:

- **Expressiveness**: For a range of interesting problems, parallel algorithms can be expressed that are: (i) not much more complicated than efficient sequential algorithms designed without regard to parallelism, and (ii) less complicated than efficient parallel algorithms expressed using less-structured parallel programming models, e.g., thread libraries providing barrier synchronization.

- **Efficiency**: For a range of interesting problems, parallel programs can be written that are comparably efficient to parallel programs written using less-structured parallel programming models.

- **Automatic Parallelization**: For a range of interesting problems, parallel programs can be written that are more efficient than parallel programs that could reasonably be expected to be generated automatically from efficient sequential programs.

- **Development Costs**: Efficient parallel programs are not much more difficult to develop than efficient sequential programs, and are less difficult to develop than efficient parallel programs written using less-structured parallel programming models. This is because most reasoning, testing, and debugging can be performed in the context of the sequential semantics.

We investigate the validity of these propositions in the context of the shared-memory multiprocessor computer system and compilation system described in Chapter 5. However, the experiments are intended to yield results that are relevant beyond this one particular example platform.

6.2 One-Deep Parallel Mergesort

In this section, we describe the development of a program that implements a scalable parallel variant on the mergesort algorithm that we refer to as the one-deep parallel mergesort algorithm. The program demonstrates the use of parallelizable for-loop statements without arguments. One-deep parallel mergesort is motivated by sequential mergesort, but the algorithm could not reasonably be expected to be automatically generated from the sequential mergesort algorithm. We compare the measured performance of the one-deep parallel mergesort program to the performance limits of the traditional approach to parallelizing mergesort. One-deep parallel mergesort is an example of the class of one-deep parallel divide-and-conquer algorithms [128].

6.2.1 Program Specification

The specification of the one-deep parallel mergesort program is given in Program 6.1. The Parallel_Mergesort procedure takes the Data array as input and returns the Result array as output. The output value of Result is the components of Data sorted into ascending order, and the output value of Data is not specified. The component type of the arrays can be any type on which ordering operators are defined. Num_Threads specifies the number of parallel threads to be used in the algorithm.

```
type Elements is array (Integer range <>) of Element;

procedure Parallel_Mergesort (
      Data        : in out Elements;
      Result      :    out Elements;
      Num_Threads : in     Positive  );
--| requires
--|     Data'Range = Result'Range and
--|     2*Num_Threads*Num_Threads ≤ Data'Length.
--| ensures
--|     Ascending(Result) and Permutation(Result, in Data).
```

Program 6.1: Specification of one-deep parallel mergesort.

6.2.2 Traditional Parallel Mergesort Algorithm

The standard sequential mergesort algorithm [73, section 5.2.4][107, chapter 12] is one of the most well-known methods of sorting and is a canonical divide-and-conquer algorithm. Sequential mergesort of an array operates as follows:

1. The input array is trivially split into a fixed number of subarrays (usually two or three) that differ in length by at most one element.

2. Each subarray is sorted using the sequential mergesort algorithm.

3. The sorted subarrays are merged to give a sorted output array.

The base case occurs for arrays with zero or one elements. The algorithm can be implemented in a top-down manner using recursion or a bottom-up manner using iteration. In both cases, a temporary storage array of the same size as the data array is required for efficient merging.

The traditional approach to parallelizing the mergesort algorithm is to sort the subarrays in parallel with each other at the outer levels of problem division. However, the maximum possible parallel speedup for sorting N elements on P processors with this approach (using two-way split and merge) is:

$$\frac{Plog_2N}{2P - 2 + log_2(\frac{N}{P})} \tag{6.1}$$

because the merge sequentially follows the parallel sorting at each level of problem division. The parallel speedup limit to traditional parallel mergesort is $\frac{1}{2}log_2N$, regardless of the number of processors that are used.

6.2.3 One-Deep Parallel Mergesort Algorithm

The one-deep parallel mergesort algorithm is a variant on the split-and-merge sorting strategy that allows considerably better parallel speedup than the traditional parallel mergesort algorithm. The one-deep parallel mergesort algorithm is presented in Program 6.2. The

```
type Elements is array (Integer range <>) of Element;
type Indices  is array (Integer range <>) of Integer;

function Split (First, Last, I : Integer; N : Positive) Return Integer is
begin
   return First + Integer(Float(Last − First + 1)*(Float(I)/Float(Num_Threads)));
end Split;

procedure Parallel_Multiway_Merge (
      Data   : in     Elements;
      P      : in     Positive;
      D      : in     Indices;
      Result :    out Elements ) is
begin
   . . .
end Parallel_Multiway_Merge;

procedure Parallel_Mergesort (
      Data        : in out Elements;
      Result      :    out Elements;
      Num_Threads : in     Positive   ) is

   D : Indices (0 .. Num_Threads);

begin
   for I in 0 .. Num_Threads loop
      D(I) := Split(Data'First, Data'Last, I, Num_Threads);
   end loop;
   pragma Parallelizable_Loop;
   for I in 0 .. Num_Threads − 1 loop
      Sequential_Quicksort(Data(D(I) .. D(I + 1) − 1));
   end loop;
   Parallel_Multiway_Merge(Data, Num_Threads, D, Result);
end Parallel_Mergesort;
```

Program 6.2: One-deep parallel mergesort.

complete text of the program is given in Appendix B.1. An example is shown in Figure 6.1. One-deep parallel mergesort of an array operates as follows:

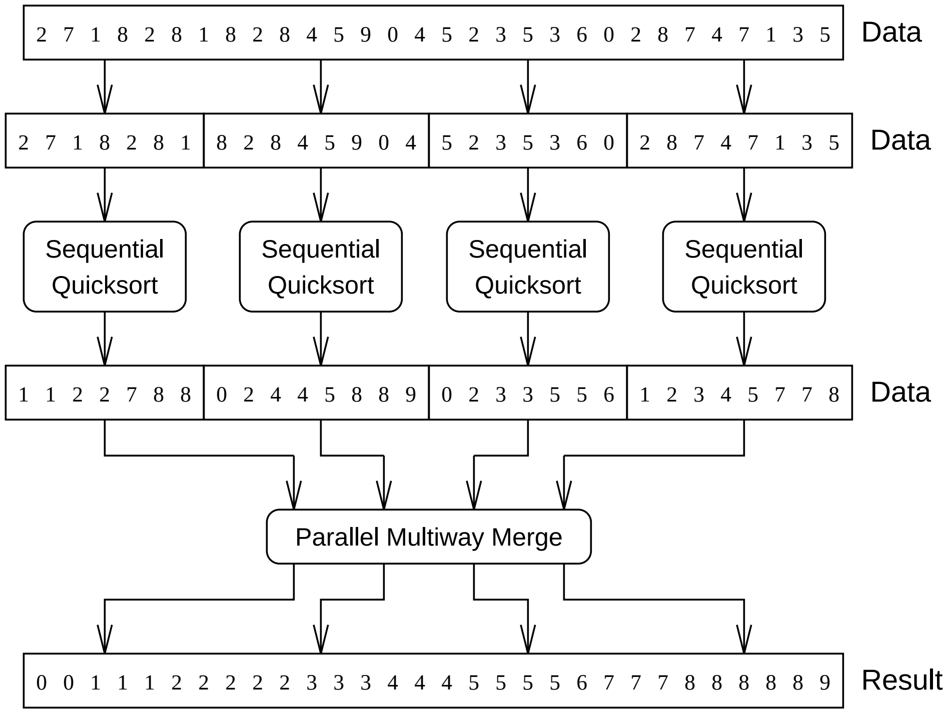


Figure 6.1: An example of one-deep parallel mergesort with four parallel threads.

1. The input array is trivially split into a user-specified number of subarrays (usually one for each processor) that differ in length by at most one element.

2. The subarrays are sorted in parallel with each other using an efficient sequential sorting algorithm (e.g., quicksort).

3. The sorted subarrays are merged using a scalable parallel multiway merge algorithm to give a sorted output array.

The key differences between this algorithm and the traditional parallel mergesort algorithm are: (i) the degree of splitting and merging is variable instead of fixed, (ii) there is only one level of problem division, and (iii) the merge is a scalable parallel algorithm. The one-deep parallel mergesort algorithm is also known as the PSRS (Parallel Sorting by Regular Sampling) algorithm [29][114].

6.2.4 Parallel Multiway Merge Algorithm

Most of the ingenuity of the one-deep parallel mergesort algorithm is in the efficient and scalable parallel multiway merge algorithm. The parallel multiway merge algorithm is presented in Program 6.3. The algorithm for a parallel P-way merge operates as follows:

1. A sequence of P evenly-spaced pairs of local pivot elements are chosen from each of the P sorted subarrays of Data.

2. The local pivot elements are sorted into ascending order.

3. A sequence of P evenly-spaced global pivot elements are chosen from the sorted local pivot elements.

4. The P sorted subarrays of Data are each partitioned into P segments by searching for the positions of the P global pivot elements in each of the subarrays.

5. For each of the P global pivot elements, the total number of elements in the corresponding segments of the P subarrays is counted.

6. In parallel, for each of the P global pivot elements, the corresponding segments of the P subarrays of Data are merged into their place in Result, using a sequential multiway merge.

For large data length and practical values of P, only the final stage takes a significant amount of time. Therefore, although several other stages can be parallelized, only the final stage is actually parallelized.

```
type Segment  is record First, Last : Integer; end record;
type Segments is array (Integer range <>) of Segment;

procedure Parallel_Multiway_Merge (
        Data   : in     Elements;
        P      : in     Positive;
        D      : in     Indices;
        Result :    out Elements ) is

    Pivot   : Elements (0 .. 2*P*P − 1);
    Index   : Integer range Data'First − 1 .. Data'Last + 1;
    Segment : array (0 .. P − 1) of Segments (0 .. P − 1);
    R       : Indices (0 .. P);

begin
    for I in 0 .. P − 1 loop
        Pivot(2*P*I) := Data(D(I));
        for J in 1 .. P − 1 loop
            Index := Split(D(I), D(I + 1) − 1, J, P);
            Pivot(2*P*I + 2*J) := Data(Index);
            Pivot(2*P*I + 2*J − 1) := Data(Index − 1);
        end loop;
        Pivot(2*P*(I + 1) − 1) := Data(D(I + 1) − 1);
    end loop;
    Sequential_Quicksort(Pivot);
    for I in 0 .. P − 1 loop
        Pivot(I) := Pivot(Split(Pivot'First, Pivot'Last, I, P));
    end loop;
    for I in 0 .. P − 1 loop
        Segment(0)(I).First := D(I);
        for J in 1 .. P − 1 loop
            Segment(J)(I).First := Search(Data(D(I) .. D(I + 1) − 1), Pivot(J));
            Segment(J − 1)(I).Last := Segment(J)(I).First − 1;
        end loop;
        Segment(P − 1)(I).Last := D(I + 1) − 1;
    end loop;
    R(0) := Result'First;
    for I in 1 .. P − 1 loop
        R(I) := R(I − 1);
        for J in 0 .. P − 1 loop
            R(I) := R(I) + Segment(I − 1)(J).Last − Segment(I − 1)(J).First + 1;
        end loop;
    end loop;
    R(P) := Result'Last + 1;
    pragma Parallelizable_Loop;
    for I in 0 .. P − 1 loop
        Sequential_Multiway_Merge(Data, P, Segment(I), Result(R(I) .. R(I + 1) − 1));
    end loop;
end Parallel_Multiway_Merge;
```

Program 6.3: Parallel multiway merge.

6.2.5 Performance Measurements

Figure 6.2 gives the measured speedup of the one-deep parallel mergesort program over an efficient sequential quicksort program. The measurements are for sorting arrays with 2.5 million and 5 million integer elements on 1 to 32 processors. In each case, the number of parallel threads is equal to the number of processors that are used, as this was found to give the best speedup. The sequential quicksort algorithm that is used for sorting the subarrays and speedup comparison is adapted from the program given by Press et al. [98, section 8.2], which uses many of the optimizations described by Sedgewick [106].

The measurements show that one-deep parallel mergesort is an effective parallel sorting algorithm. The algorithm is only marginally slower than standard quicksort when executed on one processor and produces increasing speedup with increasing numbers of processors. On 32 processors, the speedup is 16-fold for 2.5 million elements and 22-fold for 5 million elements. In comparison, the speedup limit for the traditional parallel mergesort algorithm on 32 processors is 8.7-fold for 2.5 million elements and 9-fold for 5 million elements (computed using Equation 6.1), and the achievable speedup would be somewhat less.

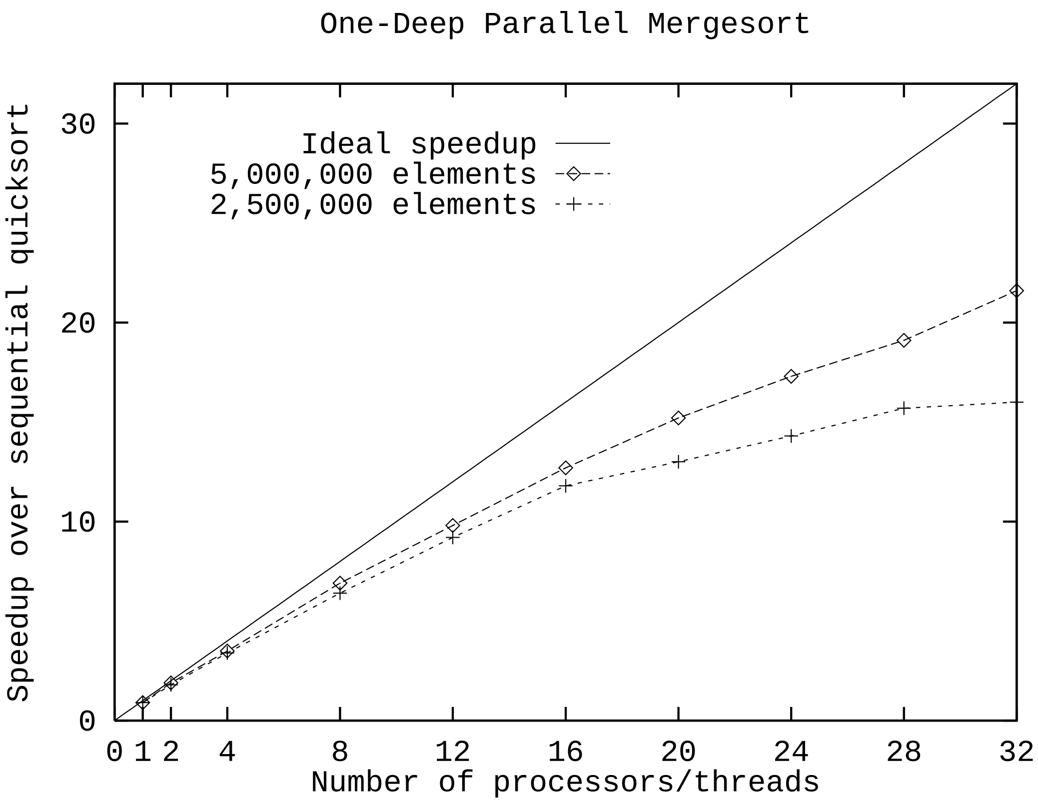
Speedup increases with increasing data length because the overheads of the algorithm become less significant compared to the increasing amount of parallelizable computation. The arrays that we sort in these experiments are relatively small compared to the total memory of the computer system, and we would expect to see even greater speedup for larger arrays. A detailed breakdown and analysis of performance is contained in earlier work by the author of this thesis [128]. This analysis indicates that algorithm overheads (e.g., uneven data partitioning and increasing computation costs) rather than system-specific factors (e.g., memory contention) and implementation overheads (e.g., thread creation and synchronization costs) are the primary limitations to speedup.

Our performance measurements are consistent with those of Shi and Schaeffer [114], who show that the one-deep parallel mergesort algorithm is among the most efficient parallel sorting algorithms currently known, for both shared-memory and distributed-memory machines. They also show that the algorithm is asymptotically optimal ($O(\frac{N}{P} logN)$ execution time) for sorting N elements on P processors with $N \geq P^3$.

6.2.6 Experimental Analysis

The development of the one-deep parallel mergesort program supports our experimental propositions as follows:

- **Expressiveness**:



Number of processors	2,500,000 elements		5,000,000 elements	
	Time	Speedup	Time	Speedup
sequential	14.59	—	31.13	—
1	16.22	0.9	33.57	0.9
2	8.30	1.8	16.73	1.9
4	4.29	3.4	8.87	3.5
8	2.29	6.4	4.51	6.9
12	1.59	9.2	3.19	9.8
16	1.24	11.8	2.46	12.7
20	1.12	13.0	2.05	15.2
24	1.02	14.3	1.80	17.3
28	0.93	15.7	1.63	19.1
32	0.91	16.0	1.44	21.6

All times are in seconds.

Figure 6.2: Speedup of one-deep parallel mergesort over sequential quicksort.

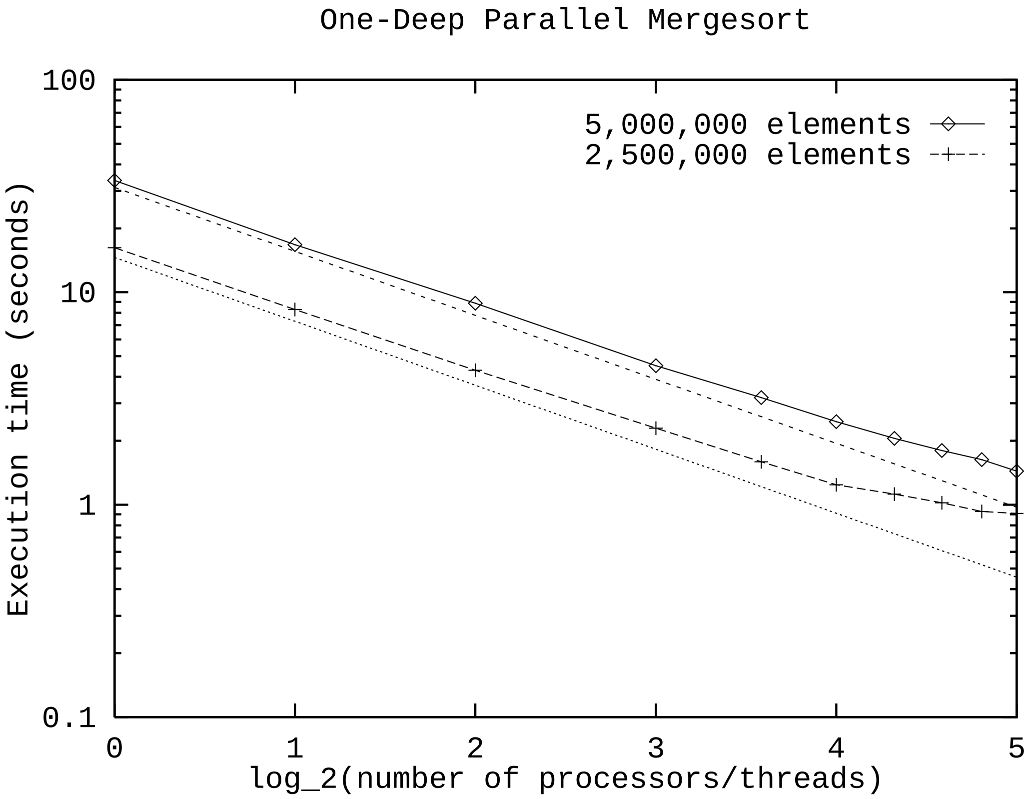


Figure 6.3: Execution time of one-deep parallel mergesort compared to ideal speedup of sequential quicksort.

At the outer level, the one-deep parallel mergesort algorithm is not much more complicated than the sequential mergesort algorithm. The major difference is that the parallelized merge algorithm is more complicated than a sequential merge algorithm. The one-deep parallel mergesort algorithm is elegantly expressed using parallelizable for-loop statements, and would be more complicated if it were expressed using less-structured parallel programming constructs, e.g., threads with barrier synchronization.

- **Efficiency**:

  The one-deep parallel mergesort program delivers good parallel speedup, increasing with number of processors and problem size. Implementation overheads do not appear to be a major limitation to speedup. Performance is significantly better than that which is possible for the traditional parallel mergesort algorithm. We do not know of a more efficient parallel sorting algorithm.

- **Automatic Parallelization**:

  The one-deep parallel mergesort algorithm could not reasonably be expected to be generated automatically from an efficient sequential mergesort algorithm. A high-level understanding of the purpose of the algorithm is required to recognize that the variable-depth split-and-merge strategy can be replaced by an equivalent fixed-depth split-and-merge strategy. Entirely new data structures and algorithms are introduced into the program.

- **Development Costs**:

  Reasoning, testing, and debugging in the context of the sequential semantics make development of the one-deep parallel mergesort program essentially no more difficult than development of a sequential program. Development would be more error-prone in a less-structured parallel programming model.

Many other divide-and-conquer algorithms are likely to be amenable to efficient parallelization using the one-deep parallel divide-and-conquer strategy. Splitting a problem into small parts and independently solving those parts in parallel leads to good cache behavior and low synchronization costs. In other work by the author of this thesis [128], we present a one-deep parallel quicksort algorithm with very similar performance to the one-deep parallel mergesort algorithm. In unpublished work, one-deep parallel divide-and-conquer algorithms have been developed for a number of other problems including the convex hull, Manhattan skyline, and closest pair problems.

6.3 The Paraffins Problem

In this section, we describe the development of a parallel program to generate the chemical structure of all paraffin molecules up to a given size. This program demonstrates the use of parallelizable for-loop statements with arguments and nested parallelizable for-loop statements. For this problem, effective parallelism requires small but significant modifications to the sequential algorithm that could not reasonably be expected to be generated automatically. Although we do not have performance measurements, to give some indication of performance limits, we compare the granularity of our program to the granularity of the straightforward approach to parallelizing the sequential algorithm.

6.3.1 Problem Description

The Paraffins problem is to output the chemical structure of all paraffin molecules with size $i \leq n$, for a given value of n, without repetition and in order of increasing size. The chemical formula for paraffin molecules is C_iH_{2i+2}. Isomers but not duplicates are to be included. Duplicates are molecules that are identical except for the ordering of bonds. Isomers are molecules that have the same chemical formula but are not duplicates. Figure 6.4 shows all the distinct paraffins of size 1, 2, 3, and 4. Figure 6.5 shows some duplicate paraffins of size 4.

Paraffin molecules of size k can be constructed from radical molecules of size less than or equal to $k/2$. A radical is a molecule with chemical formula C_iH_{2i+1}, i.e., a paraffin with one hydrogen atom removed. Figure 6.6 shows all the distinct radicals of size 0, 1, 2, and 3. Except for different bond orderings, every paraffin of size k has a unique representation as either:

1. a bond-centered paraffin: two radicals of size exactly $k/2$ bonded together, or

2. a carbon-centered paraffin: a carbon atom bonded to four radicals, each of size less than $k/2$, with combined size $k-1$.

Figure 6.7 shows examples of bond-centered and carbon-centered paraffins. The existence and uniqueness of this representation of paraffins follows from Knuth's theorems regarding enumeration of trees [72, section 2.3.4.4].

The Paraffins problem is discussed by Turner [131] and is one of the Salishan problems [37]. The Salishan problems are a set of problems proposed at the 1988 Salishan High-Speed Computing Conference as a standard by which to compare parallel programming notations. The original solutions were presented in Ada, C*, Haskell, Id, Occam,

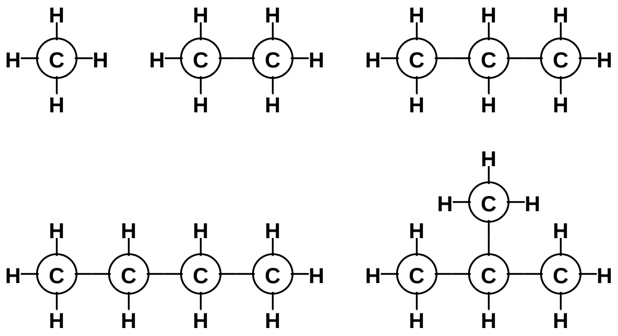


Figure 6.4: All distinct paraffins of size 1, 2, 3, and 4.

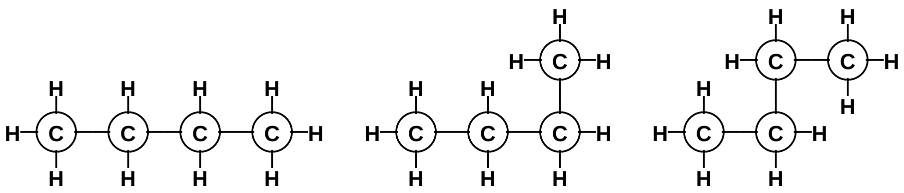


Figure 6.5: Some duplicate paraffins of size 4.

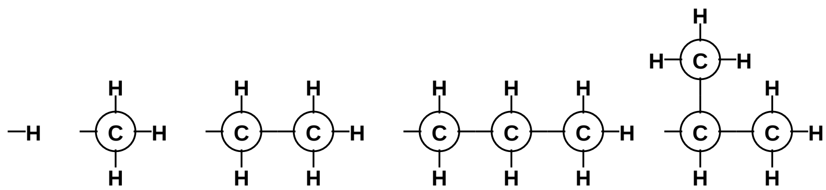


Figure 6.6: All distinct radicals of size 0, 1, 2, and 3.

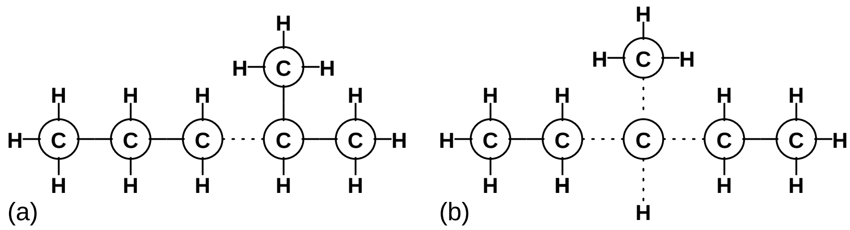


Figure 6.7: (a) An example of a bond-centered paraffin of size 6. (b) An example of a carbon-centered paraffin of size 6.

PCN, Sisal, and Scheme. Solutions to the Salishan problems in CC++ were given in earlier work by the author of this thesis [125].

6.3.2 Program Specification

The specification of a program to solve the Paraffins problem is given in Program 6.4. The

```
type Radical_Kind is (Hydrogen, Carboniferous);
type Radical;
type Radical_Pointer is access all Radical;
type Radical_Pointers is array (Integer range <>) of Radical_Pointer;
type Radical is
    record
        Kind  : Radical_Kind;
        Bonds : Radical_Pointers (1 .. 3);
    end record;
type Radical_Array is array (Integer range <>) of aliased Radical;
type Radical_Array_Pointer is access Radical_Array;
type Radical_Array_Pointers is array (Natural range <>) of Radical_Array_Pointer;

type Paraffin_Kind is (Bond_Centered, Carbon_Centered);
type Paraffin is
    record
        Kind  : Paraffin_Kind;
        Bonds : Radical_Pointers (1 .. 4);
    end record;
type Paraffin_Array is array (Integer range <>) of Paraffin;
type Paraffin_Array_Pointer is access Paraffin_Array;
type Paraffin_Array_Pointers is array (Positive range <>) of Paraffin_Array_Pointer;

procedure Generate_Paraffins (
        Radicals :    out Radical_Array_Pointers;
        Paraffins :   out Paraffin_Array_Pointers);
--| requires
--|     Radicals'First = 0 and Radicals'Last = Paraffins'Last/2.
--| ensures
--|     for all R in Radicals'Range : All_Radicals_Of_Size(R, Radicals) and
--|     for all P in Paraffins'Range : All_Paraffins_Of_Size(P, Paraffins).
```

Program 6.4: Specification of the Paraffins problem.

Generate_Paraffins procedure generates the Radicals array and the Paraffins array as output. The array bounds determine the sizes of the radicals and paraffins that are generated. The output value of each element of the arrays is a pointer to a dynamically-allocated array of all the radicals or paraffins of one size. A radical is either a single hydrogen atom or a carboniferous radical represented as an array of pointers to three smaller radicals. A bond-centered paraffin is represented as an array of pointers to two radicals, and a carbon-centered paraffin is represented as an array of pointers to four radicals.

6.3.3 Sequential Algorithm

A sequential algorithm to solve the Paraffins problem is presented in Program 6.5. The complete text of the program is given in Appendix B.2.1. The algorithm generates the radicals less than or equal to half the size of the largest paraffin, then generates the paraffins from the radicals. The first step in generating the radicals or paraffins of a given size is to compute the number of molecules of that size, so that the array can be dynamically allocated to store the molecules. Computation of the number of radicals or paraffins of a given size takes a tiny amount of time compared to the actual generation of the molecules.

A sequential algorithm to generate paraffins of a given size from radicals is presented in Program 6.6. The algorithm generates all the paraffins of size k by generating the bond-centered paraffins of size k from the radicals of size $k/2$ (if k is even), then generating the carbon-centered paraffins of size k from all the distinct combinations of four radical sizes less than $k/2$ that total to $k-1$. To prevent the generation of duplicate paraffins, different permutations of the same combination of radical sizes are not considered. The sequential algorithm for generating radicals of a given size from smaller radicals is similar in structure.

Sequential algorithms to generate bond-centered paraffins from pairs of radicals of a given size and to generate carbon-centered paraffins from radicals of four given sizes are presented in Program 6.7. The algorithms generate paraffins from all the distinct combinations of radicals of the given sizes. To prevent the generation of duplicate paraffins, different permutations of the same combination of radicals are not considered.

6.3.4 Parallel Performance Issues

On the face of it, the sequential algorithm to solve the Paraffins problem appears to be relatively simple to parallelize. We observe the following:

1. The generation of paraffins of one size is independent of the generation of paraffins of the other sizes, therefore all the different sizes of paraffins can be generated in parallel with each other.

2. The generation of paraffins of a given size does not necessarily require all the different sizes of radicals, therefore paraffins and radicals can generated in parallel with each other to some degree (with appropriate synchronization).

```
function Num_Radicals (
        Size     : Positive;
        Radicals : Radical_Array_Pointers) return Positive is
begin
   . . .
end Num_Radicals;

function Num_Paraffins (
        Size     : Positive;
        Radicals : Radical_Array_Pointers) return Positive is
begin
   . . .
end Num_Paraffins;

procedure Generate_Radicals_Of_Size (
        Result   :     out Radical_Array_Pointer;
        Size     : in      Natural;
        Radicals : in      Radical_Array_Pointers) is
   Length : Positive;
begin
   Length := Num_Radicals(Size, Radicals);
   Result := new Radical_Array (1 .. Length);
   . . .
end Generate_Radicals_Of_Size;

procedure Generate_Paraffins_Of_Size (
        Result   :     out Paraffin_Array_Pointer;
        Size     : in      Positive;
        Radicals : in      Radical_Array_Pointers) is
   Length : Positive;
begin
   Length := Num_Paraffins(Size, Radicals);
   Result := new Paraffin_Array (1 .. Length);
   . . .
end Generate_Paraffins_Of_Size;

procedure Generate_Paraffins (
        Radicals :     out Radical_Array_Pointers;
        Paraffins :    out Paraffin_Array_Pointers) is
begin
   for R in Radicals'Range loop
        Generate_Radicals_Of_Size(Radicals(R), R, Radicals(0 .. R − 1));
   end loop;
   for P in Paraffins'Range loop
        Generate_Paraffins_Of_Size(Paraffins(P), P, Radicals(0 .. P/2));
   end loop;
end Generate_Paraffins;
```

Program 6.5: Sequential algorithm to solve the Paraffins problem.

```
procedure Generate_Bond_Centered_Paraffins (
        Result   : in     Paraffin_Array_Pointer;
        Last     : in out Natural;
        Radicals : in     Radical_Array_Pointer ) is
begin
    ...
end Generate_Bond_Centered_Paraffins;

procedure Generate_Carbon_Centered_Paraffins (
        Result                         : in     Paraffin_Array_Pointer;
        Last                           : in out Natural;
        Size_1, Size_2, Size_3, Size_4 : in     Natural;
        Radicals                       : in     Radical_Array_Pointers) is
begin
    ...
end Generate_Carbon_Centered_Paraffins;

procedure Generate_Paraffins_Of_Size (
        Result   :    out Paraffin_Array_Pointer;
        Size     : in     Positive;
        Radicals : in     Radical_Array_Pointers) is

    Length : Positive;
    Last   : Natural;

begin
    Length := Num_Paraffins(Size, Radicals);
    Result := new Paraffin_Array (1 .. Length);
    Last := 0;
    if Size mod 2 = 0 then
        Generate_Bond_Centered_Paraffins(Result, Last, Radicals(Size/2));
    end if;
    for I in (Size + 2)/4 .. (Size − 1)/2 loop
        for J in (Size + 1 − I)/3 .. Min(I, Size − 1 − I) loop
            for K in (Size − I − J)/2 .. Min(J, Size − 1 − I − J) loop
                Generate_Carbon_Centered_Paraffins(
                    Result, Last, Size − 1 − I − J − K, K, J, I, Radicals);
            end loop;
        end loop;
    end loop;
end Generate_Paraffins_Of_Size;
```

Program 6.6: Sequential algorithm to generate paraffins of a given size.

```
function Last_Index (
        Outer_Size, This_Size   : Natural;
        Outer_Index, This_Last : Integer  ) return Integer is
begin
    if Outer_Size = This_Size then
        return Outer_Index;
    else
        return This_Last;
    end if;
end Last_Index;

procedure Generate_Bond_Centered_Paraffins (
        Result   : in      Paraffin_Array_Pointer;
        Last     : in out Natural;
        Radicals : in      Radical_Array_Pointer ) is
begin
    for I in 1 .. Radicals'Last loop
        for J in 1 .. I loop
            Last := Last + 1;
            Result(Last) := (Bond_Centered,
                (Radicals(I)'Access, Radicals(J)'Access, null, null));
        end loop;
    end loop;
end Generate_Bond_Centered_Paraffins;

procedure Generate_Carbon_Centered_Paraffins (
        Result                         : in      Paraffin_Array_Pointer;
        Last                           : in out Natural;
        Size_1, Size_2, Size_3, Size_4 : in      Natural;
        Radicals                       : in      Radical_Array_Pointers) is
begin
    for I in 1 .. Radicals(Size_1)'Last loop
        for J in 1 .. Last_Index(Size_1, Size_2, I, Radicals(Size_2)'Last) loop
            for K in 1 .. Last_Index(Size_2, Size_3, J, Radicals(Size_3)'Last) loop
                for L in 1 .. Last_Index(Size_3, Size_4, K, Radicals(Size_4)'Last) loop
                    Last := Last + 1;
                    Result(Last) := (Carbon_Centered,
                        (Radicals(Size_1)(I)'Access, Radicals(Size_2)(J)'Access,
                         Radicals(Size_3)(K)'Access, Radicals(Size_4)(L)'Access));
                end loop;
            end loop;
        end loop;
    end loop;
end Generate_Carbon_Centered_Paraffins;
```

Program 6.7: Sequential algorithms to generate bond-centered and carbon-centered paraffins.

3. Similarly, radicals of different sizes can be generated in parallel with each other to some degree (with appropriate synchronization).

Based on these observations, a straightforward parallelized algorithm to solve the Paraffins problem is presented in Program 6.8. In this algorithm, all the different sizes of radicals and

```
...
pragma Single_Assignment(Radical_Array_Pointer);
...

procedure Generate_Paraffins (
        Radicals :    out Radical_Array_Pointers;
        Paraffins :   out Paraffin_Array_Pointers) is
begin
    pragma Parallelizable_Sequence;
    pragma Parallelizable_Loop;
    for R in Radicals'Range loop
        Generate_Radicals_Of_Size(Radicals(R), R, Radicals(0 .. R − 1));
    end loop;
    pragma Parallelizable_Loop;
    for P in Paraffins'Range loop
        Generate_Paraffins_Of_Size(Paraffins(P), P, Radicals(0 .. P/2));
    end loop;
end Generate_Paraffins;
```

Program 6.8: Straightforward parallelized algorithm to solve the Paraffins problem.

paraffins are generated in parallel with each other. Generation and evaluation of radicals is synchronized by making Radical_Array_Pointer a single-assignment type (and by a few trivial changes to Generate_Radicals_Of_Size). Essentially, this is the algorithm presented in most of the original solutions (and our own CC++ solutions) to the Salishan problems.

Although this algorithm executes as a reasonable number of parallel threads, the parallel speedup it can yield is negligible, because most of the computation occurs in the generation of the paraffins of the largest size. For example, Table 6.1 gives the numbers of radicals and paraffins in the generation of paraffins up to size 24. Radicals make up only 0.01 percent of the total number of molecules generated (and this percentage decreases as the problem size increases). Paraffins of the largest size make up 60 percent of the total number of molecules generated (and this percentage increases as the problem size increases). Therefore, the maximum possible speedup for generating paraffins up to size 24 using the straightforward parallelized algorithm is $1\frac{2}{3}$-fold, regardless of the number of processors that are used. Parallel performance is dependent on effective parallelization of the generation of the largest

	Radicals		Paraffins	
Size	Number	Total	Number	Total
0	1	1	—	—
1	1	2	1	1
2	1	3	1	2
3	2	5	1	3
4	4	9	2	5
5	8	17	3	8
6	17	34	5	13
7	39	73	9	22
8	89	162	18	40
9	211	373	35	75
10	507	880	75	150
11	1,238	2,118	159	309
12	3,057	5,175	355	664
13	—	—	802	1,466
14	—	—	1,858	3,324
15	—	—	4,347	7,671
16	—	—	10,359	18,030
17	—	—	24,894	42,924
18	—	—	60,523	103,447
19	—	—	148,284	251,731
20	—	—	366,319	618,050
21	—	—	910,726	1,528,776
22	—	—	2,278,658	3,807,434
23	—	—	5,731,580	9,539,014
24	—	—	14,490,245	24,029,259

Table 6.1: Number of radicals of size 0 to 12 and number of paraffins of size 1 to 24.

sized paraffins, and it is not worthwhile to parallelize the generation of radicals.

6.3.5 Parallel Algorithm

An efficient parallelized algorithm to solve the Paraffins problem is presented in Program 6.9. The complete text of the program is given in Appendix B.2.2. In this algorithm, radicals

```
procedure Generate_Paraffins (
        Radicals :    out Radical_Array_Pointers;
        Paraffins :   out Paraffin_Array_Pointers) is
begin
    for R in Radicals'Range loop
        Generate_Radicals_Of_Size(Radicals(R), R, Radicals(0 .. R − 1));
    end loop;
    pragma Parallelizable_Loop(Num_Threads => 2, Pattern => On_Demand);
    for P in reverse Paraffins'Range loop
        Generate_Paraffins_Of_Size(Paraffins(P), P, Radicals(0 .. P/2));
    end loop;
end Generate_Paraffins;
```

Program 6.9: Efficient parallelized algorithm to solve the Paraffins problem.

are generated before paraffins are generated. Paraffins are generated in reverse order of size, using two parallel threads with iterations assigned to threads using the On_Demand pattern. Since the number of paraffins of a given size increases with size, reverse order helps load balancing by scheduling large computations before small computations. Since there are more paraffins of the largest size than all the other paraffins combined, there is no benefit to having more than two parallel threads at the outer level. To provide effective parallelism, paraffins of the same size must be generated in parallel.

The parallelized algorithm to generate the paraffins of a given size is presented in Program 6.10. The sequential algorithm is parallelized in three places: (i) bond-centered and carbon-centered paraffins are generated in parallel with each other, (ii) the outer two loops of the three nested loops that generate carbon-centered paraffins are parallelized, and (iii) the algorithm to generate bond-centered paraffins is parallelized, with the same number of parallel threads as the paraffin size. Interference between parallel threads is avoided by precomputing the number of bond-centered and carbon-centered paraffins and the index in Result of the carbon-centered paraffins with each combination of radical sizes. This precomputation takes a tiny amount of time compared to the actual generation of the paraffins.

The parallelized algorithm to generate bond-centered paraffins is presented in Pro-

```
...
type Indices is array (Natural range <>,
                       Natural range <>,
                       Natural range <> ) of Positive;
...

procedure Generate_Paraffins_Of_Size (
        Result   :     out Paraffin_Array_Pointer;
        Size     : in      Positive;
        Radicals : in      Radical_Array_Pointers) is

        Num_Bond_Centered   : Natural;
        Num_Carbon_Centered : Natural;
        Num_Paraffins       : Positive;
        First               : Indices ((Size + 2)/4 .. (Size − 1)/2,
                                       (Size + 3)/6 .. (Size − 1)/2,
                                       0 .. (Size − 1)/3);
begin
    Count_Bond_Centered(Size, Radicals(Size/2), Num_Bond_Centered);
    Count_Carbon_Centered(Size, Radicals, First, Num_Carbon_Centered);
    Num_Paraffins := Num_Bond_Centered + Num_Carbon_Centered;
    Result := new Paraffin_Array (1 .. Num_Paraffins);
    begin
        pragma Parallelizable_Sequence;
        if Size mod 2 = 0 then
            Generate_Bond_Centered_Paraffins(
                Result, Radicals(Size/2), Num_Threads => Size);
        end if;
        pragma Parallelizable_Loop;
        for I in (Size + 2)/4 .. (Size − 1)/2 loop
            pragma Parallelizable_Loop;
            for J in (Size + 1 − I)/3 .. Min(I, Size − 1 − I) loop
                for K in (Size − I − J)/2 .. Min(J, Size − 1 − I − J) loop
                    Generate_Carbon_Centered_Paraffins(
                        Result, Num_Bond_Centered + First(I, J, K),
                        Size − 1 − I − J − K, K, J, I, Radicals);
                end loop;
            end loop;
        end loop;
    end;
end Generate_Paraffins_Of_Size;
```

Program 6.10: Parallelized algorithm to generate paraffins of a given size.

gram 6.10. The outer loop is parallelized with iterations assigned to parallel threads using

```
procedure Generate_Bond_Centered_Paraffins (
        Result      : in    Paraffin_Array_Pointer;
        Radicals    : in    Radical_Array_Pointer;
        Num_Threads : in    Positive              ) is
begin
    pragma Parallelizable_Loop(Num_Threads, Pattern => Cyclic);
    for I in 1 .. Radicals'Last loop
        declare
            Base : constant Natural := (I*(I - 1))/2;
        begin
            for J in 1 .. I loop
                Result(Base + J) := (Bond_Centered,
                    (Radicals(I)'Access, Radicals(J)'Access, null, null));
            end loop;
        end;
    end loop;
end Generate_Bond_Centered_Paraffins;
```

Program 6.11: Parallelized algorithm to generate bond-centered paraffins.

the Cyclic pattern. The Cyclic pattern ensures that the computation performed by each thread is approximately equal without the overhead of the On_Demand pattern.

6.3.6 Performance Indications

Because the current release of the GNAT compilation system does not correctly implement nested parallelism on multiprocessors, we do not have parallel performance measurements for the parallelized algorithm to solve the Paraffins problem.

We can give some indication of performance possibilities by computing the granularity of the parallelism expressed by the algorithm. For generation of paraffins up to size 24, generation of the largest sized paraffin is split into 50 parallel threads. The largest number of paraffins generated by any one of these threads is 765,703, which is approximately 3.2 percent of the total number of paraffins generated. Therefore, the speedup for generating paraffins up to size 24 using the efficient parallelized algorithm could be up to 31-fold. This compares to a maximum possible speedup of $1\frac{2}{3}$-fold for the straightforward parallelized algorithm.

Further indication of the likelihood of good parallel performance is given by preliminary experiments which show that the parallel program executes insignificantly slower than the sequential program on a single processor. For generation of paraffins up to size 24, the execution time of sequential program is approximately 210 seconds and the execution time of the parallel program on a single processor is approximately 213 seconds.

6.3.7 Experimental Analysis

The development of the parallelized algorithm to solve the Paraffins problem supports our experimental propositions as follows:

- **Expressiveness**:

  The parallelized algorithm to solve the Paraffins problem is almost the same as the sequential algorithm to solve the Paraffins problem. The major difference is the precomputation of the indices of the paraffins. The parallelized algorithm is elegantly expressed using nested parallelizable sequences of statements and parallelizable for-loop statements with iterations assigned to parallel threads using the On_Demand and Cyclic patterns. The patterns of thread creation and synchronization are relatively sophisticated, and the algorithm would be more complicated if it were expressed using less-structured parallel programming constructs, e.g., threads with barrier synchronization.

- **Efficiency**:

  Although we do not have performance measurements, the parallelized program to solve the Paraffins problem expresses sufficiently many threads of sufficiently fine granularity to indicate that the program would demonstrate at least reasonable parallel speedup on a moderate number of processors. We do not know of a more efficient parallel algorithm to solve the Paraffins problem.

- **Automatic Parallelization**:

  The efficient parallelized algorithm could not reasonably be expected to be generated automatically from an efficient sequential algorithm to solve the Paraffins problem. A high-level understanding of the purpose of the algorithm is required to recognize that the indices of the paraffins can be precomputed to allow parallel generation of paraffins of the same size. Entirely new data structures and algorithms are introduced to the program. In addition, automatic program parallelization is difficult for programs that involve pointers to dynamically-allocated data structures, because of potential aliasing.

- **Development Costs**:

> Reasoning, testing, and debugging in the context of the sequential semantics make development of the efficient parallelized program to solve the Paraffins problem essentially no more difficult than development of a sequential program. Development would be more error-prone in a less-structured parallel programming model, particularly because of the sophisticated patterns of thread creation and synchronization.

All aspects of the benefits of a high-level parallel programming model with direct control of parallelism are increased for a problem such as the Paraffins problem in which nested parallelism and irregular patterns of thread creation are required and in which load balancing is non-trivial.

Chapter 7

Experiments Using Single-Assignment Types

In this chapter, we describe parallel programming experiments designed to investigate the use of single-assignment types for synchronization between parallel threads of control. The experiments are performed in the context of the hardware and software platforms described in Chapter 5.

7.1 Experimental Goals

The programming experiments that we describe in this chapter investigate parallel programming using parallelizable sequences of statements, parallelizable for-loop statements, and single-assignment types. The goal of these experiments is to evaluate the following propositions with respect to this programming model:

- **Expressiveness**: For a range of interesting problems, parallel algorithms can be expressed that are: (i) not much more complicated than efficient sequential algorithms designed without regard to parallelism, and (ii) less complicated than efficient parallel algorithms expressed using less-structured parallel programming models, e.g., thread libraries providing barrier and lock synchronization.

- **Efficiency**: For a range of interesting problems, parallel programs can be written that are: (i) more efficient than parallel programs that can be written using the same model without single-assignment types, and (ii) comparably efficient to parallel programs written using less-structured parallel programming models.

- **Automatic Parallelization**: For a range of interesting problems, parallel programs can be written that are more efficient than parallel programs that could reasonably be expected to be generated automatically from efficient sequential programs.

- **Development Costs**: Efficient parallel programs are not much more difficult to develop than efficient sequential programs, and are less difficult to develop than efficient parallel programs written using less-structured parallel programming models. This is because most reasoning, testing, and debugging can be performed in the context of the sequential semantics.

We investigate the validity of these propositions in the context of the shared memory multiprocessor computer system and compilation system described in Chapter 5. However, the experiments are intended to yield results that are relevant beyond this one particular example platform.

7.2 Mergesort of a Linked List

In this section, we describe the development of a parallel program to sort linked lists (with single-assignment links) into ascending order using the mergesort algorithm. The program demonstrates the use of single-assignment types in the context of dynamically-allocated data structures, and the use of the Priority pragma to improve load balancing. Although the parallel mergesort algorithm is almost identical to the sequential mergesort algorithm, the subtle synchronization in the parallel algorithm could not reasonably be expected to be generated automatically from the sequential algorithm. We compare the measured performance of parallel mergesort of linked lists with single-assignment links to the measured performance of traditional parallel mergesort of linked lists with ordinary mutable links.

7.2.1 Program Specification

The specification of the mergesort program is given in Program 7.1. The Mergesort procedure takes the Unsorted list as input and returns the Sorted list as output. The declaration of the list type is given in Section 7.2.2, and the element type of a list can be any type on which ordering operators are defined. The output value of Sorted is the elements of Unsorted sorted into ascending order, and the output value of Unsorted is not specified. Parallel_Depth specifies the recursive depth of parallel sorting and hence the number of parallel threads to be used in the algorithm.

```
type List is ... ;

procedure Mergesort (Unsorted, Sorted : in out List; Parallel_Depth : in Natural);
--| requires
--|     Closed(Unsorted) and not Closed(Sorted) and Empty(Sorted).
--| ensures
--|     Closed(Sorted) and Ascending(Sorted) and Permutation(Sorted, in Unsorted).
```

Program 7.1: Specification of parallel mergesort of a linked list.

7.2.2 Linked Lists with Single-Assignment Links

The declaration of the list type is given in Program 7.2. A list consists of Head and Tail pointers to a forward-linked list of nodes. Each node consists of an Item pointer to a block of data elements and a Next pointer to the next node in the list. Links between nodes are single-assignment pointers, and all other pointers are ordinary mutable pointers. Blocking of data reduces the amount of storage taken by the links and reduces the time spent traversing links, at the cost of an average of one half-filled block per list. Except for the single-assignment links, this is an unremarkable declaration of a linked list type, such as might be used in a sequential program. Figure 7.1 shows an example of a linked list of blocks of integers.

Subprograms are defined for writing a (pointer to a) block of elements to the tail of a list and reading a (pointer to a) block of elements from the head of a list. Subprograms are also defined for closing writing to a list and for testing whether reading has reached the end of a list. The implementation of some of these operations on lists is given in Program 7.3. In the implementation, the Head pointer points to the last node that was read, and the Tail pointer points to the last node that was written. Except for the single-assignment links, this is an unremarkable implementation of an unremarkable set of operations on a linked list type, such as might be used in a sequential program.

If the links between nodes were mutable pointers, a parallel reader and writer of a list would be erroneous, because of the parallel assignment and evaluation of nodes and links (violating restriction E.1). With single-assignment pointers as links between nodes, a parallel reader and writer of a list is not erroneous, because evaluation of an unassigned single-assignment link by a Get operation will automatically suspend until the link is assigned by a Put operation. The implementation ensures that the operations on the mutable components of nodes are synchronized by the operations on the single-assignment links. In this manner, a parallel reader and writer of a list are implicitly synchronized by operations

```
type Element is ... ;
type Elements is array (Integer range <>) of Element;

type Block (Block_Length : Positive) is
   record
      Data   : Elements (1 .. Block_Length);
      Length : Positive;
   end record;
type Block_Access is access Block;

type Node;
type Pointer is access Node;
type Single_Pointer is new Pointer;
pragma Single_Assignment(Single_Pointer);
type Node is
   record
      Item : Block_Access;
      Next : Single_Pointer;
   end record;
type List (Block_Length : Positive) is new Limited_Controlled with
   record
      Head : Pointer;
      Tail : Pointer;
   end record;

procedure Initialize (L : in out List);

procedure Finalize (L : in out List);

procedure Put (L : in out List; Item : in Block_Access);

procedure Close (L : in out List);

function End_Of_List (L : List) return Boolean;

procedure Get (L : in out List; Item : out Block_Access);

procedure Get (L : in out List; Item : out Block_Access; Past_End : out Boolean);
```

Program 7.2: Declaration of the list type.

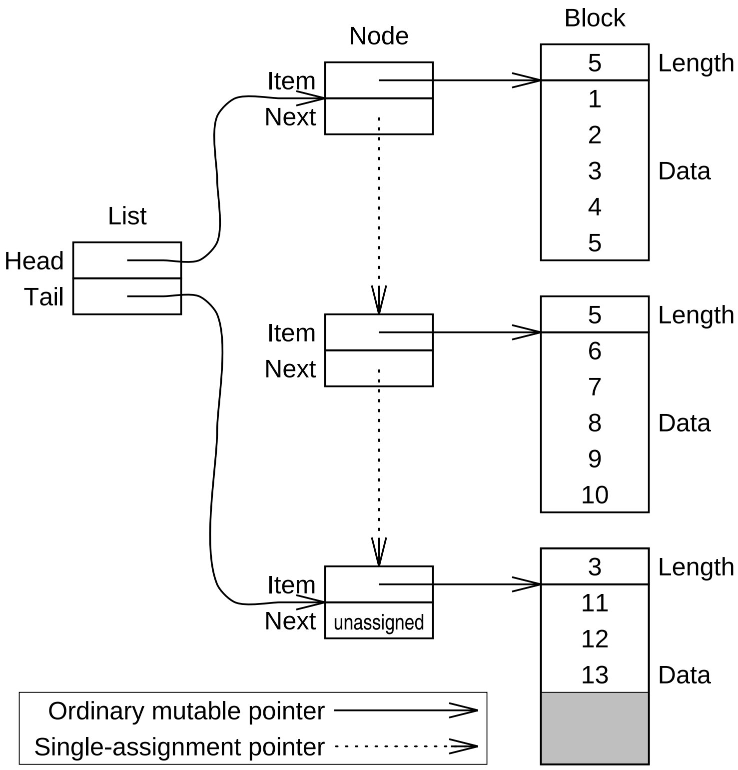


Figure 7.1: An example of a linked list of blocks of integers.

```
procedure Initialize (L : in out List) is
begin
    L.Head := new Node;
    L.Head.Item := null;
    L.Tail := L.Head;
end Initialize;

procedure Finalize (L : in out List) is
    P, Next : Pointer;
begin
    P := L.Head;
    while P /= L.Tail loop
        Next := Pointer(P.Next);
        Deallocate(P);
        P := Next;
    end loop;
    Deallocate(P);
end Finalize;

procedure Put (L : in out List; Item : in Block_Access) is
    New_Tail : Pointer;
begin
    New_Tail := new Node;
    New_Tail.Item := Item;
    L.Tail.Next := Single_Pointer(New_Tail);
    L.Tail := New_Tail;
end Put;

procedure Close (L : in out List) is
begin
    L.Tail.Next := null;
end Close;

function End_Of_List (L : List) return Boolean is
begin
    return L.Head.Next = null;
end End_Of_List;

procedure Get (L : in out List; Item : out Block_Access) is
    Old_Head : Pointer;
begin
    Old_Head := L.Head;
    L.Head := Pointer(L.Head.Next);
    Item := L.Head.Item;
    Deallocate(Old_Head);
end Get;
```

Program 7.3: Implementation of some operations on lists.

on the single-assignment links. Multiple parallel readers of a list or multiple parallel writers to a list would be erroneous due to violations of restrictions E.1 and E.2. Note that a **null** link indicates the assigned end of a closed list and is different from an unassigned link.

7.2.3 Parallel Mergesort with Mutable Links

Before we present the parallel mergesort algorithm for linked lists with single-assignment links, we investigate a traditional parallel mergesort algorithm for linked lists with mutable links. We compare the two algorithms and their performance measurements. The traditional parallel mergesort algorithm for linked lists with mutable links is presented in Program 7.4. The complete text of the program is given in Appendix B.3.3. The linked list type with mutable links is essentially identical to the linked list type with single-assignment links, except that the links between nodes are mutable pointers and there is no need for a Close operation. Traditional parallel mergesort for linked lists with mutable links operates as follows:

1. If the input list is empty, the output list is left empty.

2. If the input list consists of a single block, the block is sorted using sequential quicksort and written to the output list.

3. If the input list consists of more than one block: (i) the input list is sequentially split into two sublists that differ in length by at most one block, (ii) the two sublists are recursively sorted, then (iii) the two sorted sublists are sequentially merged to give the sorted output list. At the outer levels of recursion, the two sublists are sorted in parallel with each other.

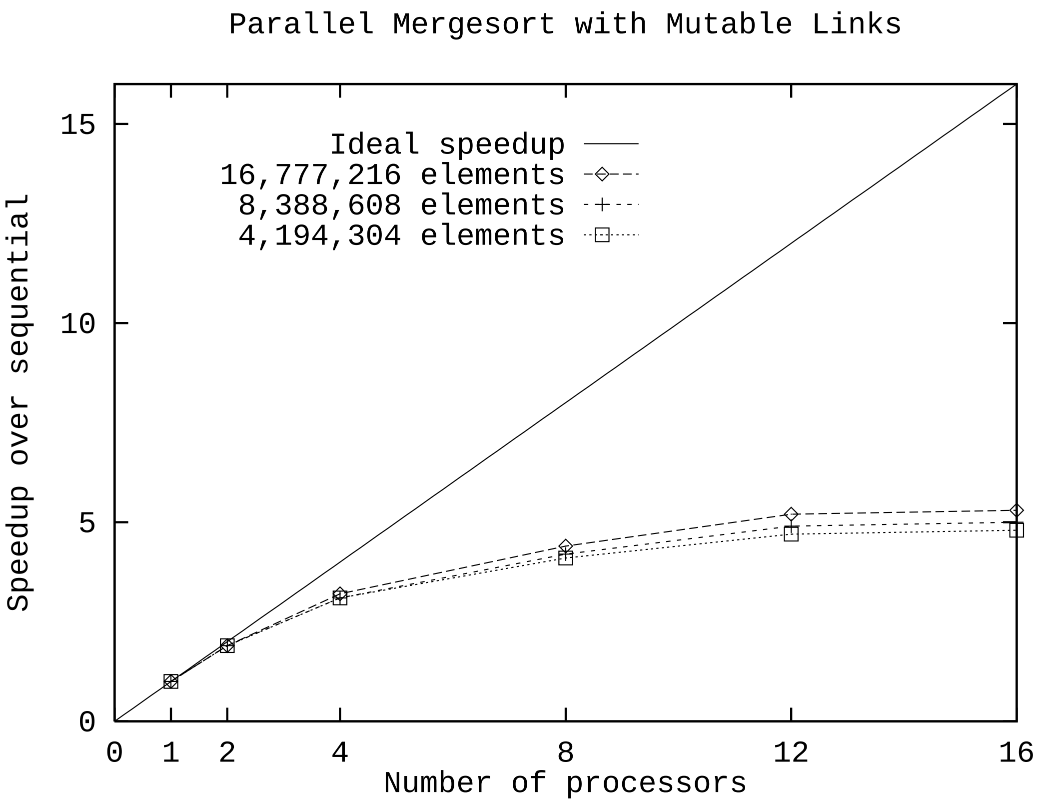
Splitting the input list into two sublists is an inexpensive operation, because only the pointers to the data blocks are copied, not the data elements. Merging the sorted sublists is an expensive operation, because the actual data elements are compared and copied to the output list. Since the links between nodes are mutable pointers, neither the split nor the merge can be executed in parallel with the recursive sorting. As discussed in Chapter 6, the parallel speedup of the traditional parallel mergesort algorithm is severely limited because the merge sequentially follows the parallel sorting.

7.2.4 Performance Measurements with Mutable Links

Figure 7.2 gives the measured speedup of the traditional parallel mergesort program over an efficient sequential mergesort program for linked lists with mutable links. The measure-

```
procedure Quicksort (Data : in out Elements) is
begin
    ...
end Quicksort;

procedure Split (Input, Left, Right : in out List) is
begin
    ...
end Split;

procedure Merge (Left, Right, Output : in out List) is
begin
    ...
end Merge;

procedure Mergesort (Unsorted, Sorted : in out List; Parallel_Depth : in Natural) is

        Block_1, Block_2      : Block_Access;
        Get_Past_End          : Boolean;
        Left, Sorted_Left     : List (Unsorted.Block_Length);
        Right, Sorted_Right : List (Unsorted.Block_Length);

begin
    Get(Unsorted, Block_1, Get_Past_End);
    if not Get_Past_End then
        Get(Unsorted, Block_2, Get_Past_End);
        if Get_Past_End then
            Quicksort(Block_1.Data(1 .. Block_1.Length));
            Put(Sorted, Block_1);
        else
            Put(Left, Block_1); Put(Right, Block_2);
            Split(Unsorted, Left, Mid, Right);
            if Parallel_Depth = 0 then
                Mergesort(Left, Sorted_Left, 0);
                Mergesort(Right, Sorted_Right, 0);
            else
                pragma Parallelizable_Sequence;
                Mergesort(Left, Sorted_Left, Parallel_Depth − 1);
                Mergesort(Right, Sorted_Right, Parallel_Depth − 1);
            end if;
            Merge(Sorted_Left, Sorted_Right, Sorted);
        end if;
    end if;
end Mergesort;
```

Program 7.4: Traditional parallel mergesort for linked lists with mutable links.



Number of processors	4,194,304 elements		8,388,608 elements		16,777,216 elements	
	Time	Speedup	Time	Speedup	Time	Speedup
sequential	84.15	—	172.88	—	363.20	—
1	83.84	1.0	172.83	1.0	363.75	1.0
2	43.76	1.9	91.21	1.9	192.81	1.9
4	27.38	3.1	56.19	3.1	113.50	3.2
8	20.37	4.1	40.85	4.2	82.85	4.4
12	17.73	4.7	34.94	4.9	70.32	5.2
16	17.68	4.8	34.31	5.0	68.08	5.3

All times are in seconds.

Figure 7.2: Speedup of traditional parallel mergesort over sequential mergesort for linked lists with mutable links (block length = 65,536 elements).

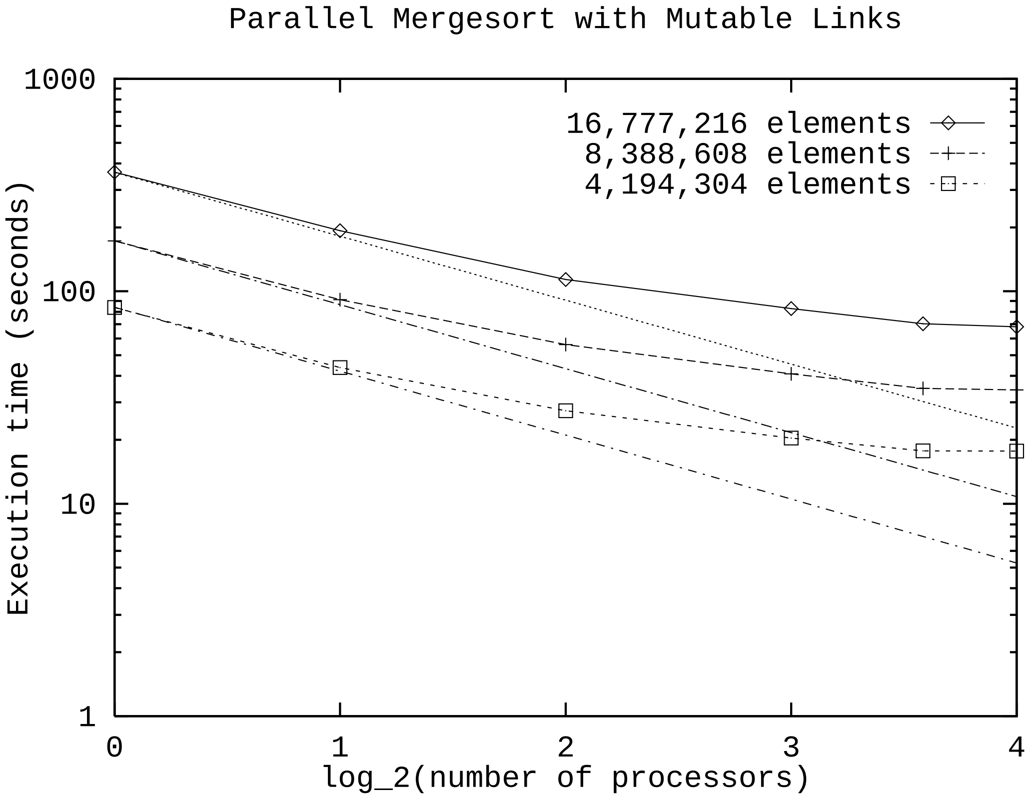


Figure 7.3: Execution time of traditional parallel mergesort compared to ideal speedup of sequential mergesort for linked lists with mutable links (block length = 65,536 elements).

ments are for sorting lists with 4,194,304 to 16,777,216 integer elements in blocks of 65,536 elements on 1 to 16 processors. In each case, the parallel depth is the smallest depth that gives at least one parallel thread per processor at the bottom level of the parallel recursion (e.g., for 8 processors, the parallel depth is 3), as this was found to give the best speedup. The sequential mergesort program that is used for speedup comparison is identical to the parallel mergesort program, except that the recursive sorting of the sublists is performed sequentially.

As expected, the measurements show that the traditional parallel mergesort algorithm for linked lists with mutable links is only effective for a small number of processors. The parallel algorithm delivers good speedup for up to four processors and very little additional speedup for more than four processors. Speedups increase very little with increasing data length. This performance pattern is consistent with Equation 6.1 and with other reported results [114].

7.2.5 Parallel Mergesort with Single-Assignment Links

The parallel mergesort algorithm for linked lists with single-assignment links is presented in Program 7.5. The complete text of the program is given in Appendix B.3.5. The parallel mergesort for linked lists with single-assignment links is essentially identical to the traditional parallel mergesort for linked lists with mutable links, except that the single-assignment links allow the merge to be performed in parallel with the recursive sorting of the sublists. (The split could also be performed in parallel with the merge and recursive sorting, but takes too little time for this to be worthwhile.) The split and merge operations are essentially unchanged from the traditional parallel mergesort for linked lists with mutable links.

Synchronization between parallel merge and sorting operations is through read operations suspending on the evaluation of unassigned links and resuming when the links are assigned values. In this manner, the network of parallel merge and sorting operations are implicitly synchronized by the production and consumption of input and output lists. The Priority pragma is used to give higher scheduling priorities to the merge operations at the outer levels of recursion, which merge larger lists. Without assignment of priorities, poor load balancing can occur as a result of the largest merge operations not beginning execution until near the end of the computation.

```
procedure Quicksort (Data : in out Elements) is
begin
    . . .
end Quicksort;

procedure Split (Input, Left, Right : in out List) is
begin
    . . .
end Split;

procedure Merge (Left, Right, Output : in out List) is
begin
    . . .
end Merge;

procedure Mergesort (Unsorted, Sorted : in out List; Parallel_Depth : in Natural) is

        Block_1, Block_2      : Block_Access;
        Get_Past_End          : Boolean;
        Left, Sorted_Left     : List (Unsorted.Block_Length);
        Right, Sorted_Right   : List (Unsorted.Block_Length);

begin
    Get(Unsorted, Block_1, Get_Past_End);
    if Get_Past_End then
        Close(Sorted);
    else
        Get(Unsorted, Block_2, Get_Past_End);
        if Get_Past_End then
            Quicksort(Block_1.Data(1 .. Block_1.Length));
            Put(Sorted, Block_1);
            Close(Sorted);
        else
            Put(Left, Block_1); Put(Right, Block_2);
            Split(Unsorted, Left, Right);
            if Parallel_Depth = 0 then
                Mergesort(Left, Sorted_Left, 0);
                Mergesort(Right, Sorted_Right, 0);
                Merge(Sorted_Left, Sorted_Right, Sorted);
            else
                pragma Parallelizable_Sequence;
                Mergesort(Left, Sorted_Left, Parallel_Depth - 1);
                Mergesort(Right, Sorted_Right, Parallel_Depth - 1);
                pragma Priority(Default_Priority + Parallel_Depth);
                Merge(Sorted_Left, Sorted_Right, Sorted);
            end if;
        end if;
    end if;
end Mergesort;
```

Program 7.5: Parallel mergesort for linked lists with single-assignment links.

7.2.6 Performance Measurements with Single-Assignment Links

Figure 7.4 gives the measured speedup of the parallel mergesort program for linked lists with single-assignment links over an efficient sequential mergesort program for linked lists with mutable links. The measurements are for sorting lists with 4,194,304 to 16,777,216 integer elements in blocks of 65,536 elements on 1 to 16 processors. In each case, the parallel depth is the smallest depth that gives at least one parallel thread per processor at the bottom level of the parallel recursion (e.g., for 8 processors, the parallel depth is 3), as this was found to give the best speedup. The sequential mergesort program that is used for speedup comparison is essentially identical to the parallel mergesort program, except that the links in the lists are mutable pointers and the recursive sorting and merging are performed sequentially.

The measurements show that the parallel mergesort program for linked lists with single-assignment links delivers considerably better speedup than the traditional parallel mergesort program for linked lists with mutable links. For sorting 16,777,216 elements on 16 processors, the execution time of parallel mergesort for linked lists with single-assignment links is 60 percent of the execution time of parallel mergesort for linked lists with mutable links. The speedup scales to increasing numbers of processors and shows greater increases with increasing data length. The additional performance is obtained without adding any complexity to the program.

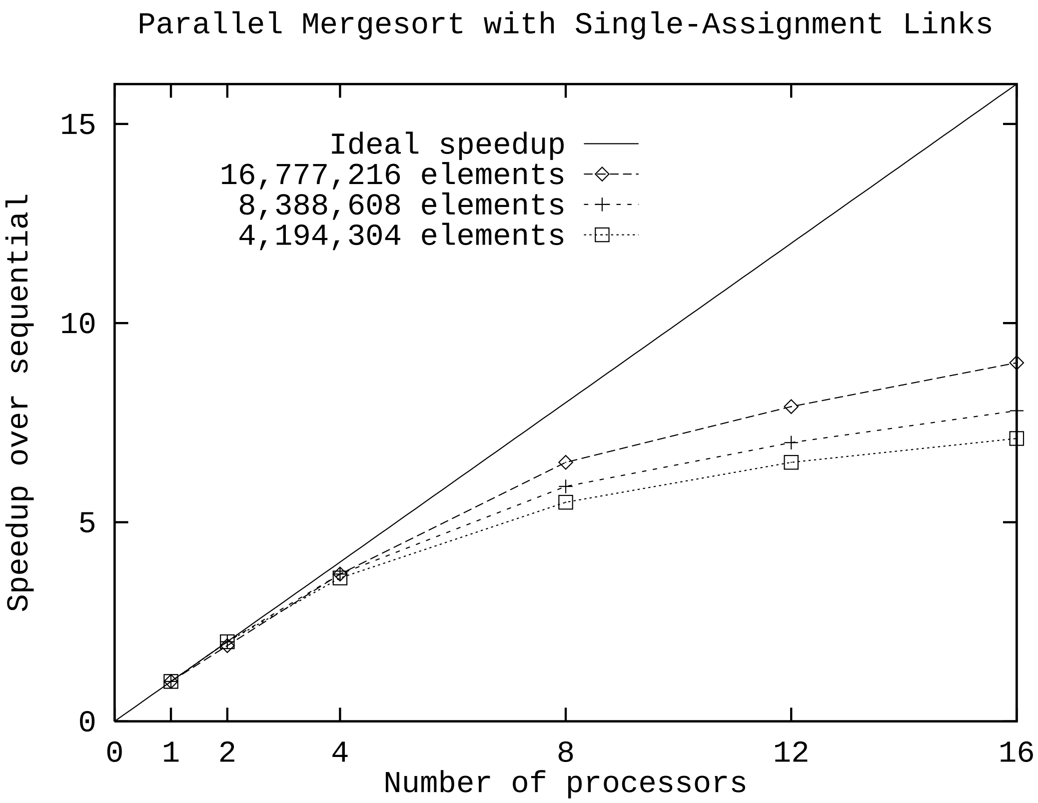
The parallel mergesort algorithm for linked lists with single-assignment links is slower and does not scale as well as the one-deep parallel mergesort algorithm for arrays presented in Chapter 6. Nonetheless, parallel mergesort of a linked list may be an appropriate parallel sorting algorithm for the following reasons: (i) parallel mergesort of a linked list requires considerably less extra data storage than one-deep parallel mergesort, (ii) if the data length cannot be determined prior to the generation of the data, a linked list may be the most appropriate data storage structure, and (iii) a linked list allows dynamic distribution of the data across the memories of several different processors.

7.2.7 Experimental Analysis

The development of the parallel mergesort program for linked lists with single-assignment links supports our experimental propositions as follows:

- **Expressiveness**:

  The parallel mergesort algorithm for linked lists with single-assignment links is essentially identical to an efficient sequential mergesort algorithm for linked lists with



Number of processors	4,194,304 elements		8,388,608 elements		16,777,216 elements	
	Time	Speedup	Time	Speedup	Time	Speedup
sequential	84.15	—	172.88	—	363.20	—
1	84.13	1.0	172.83	1.0	364.53	1.0
2	41.61	2.0	86.67	2.0	182.89	1.9
4	23.20	3.6	46.89	3.7	97.90	3.7
8	15.24	5.5	29.17	5.9	55.96	6.5
12	12.95	6.5	24.74	7.0	46.07	7.9
16	11.82	7.1	22.20	7.8	40.54	9.0

All times are in seconds.

Figure 7.4: Speedup of parallel mergesort for linked lists with single-assignment links over sequential mergesort for linked lists with mutable links (block length = 65,536 elements).

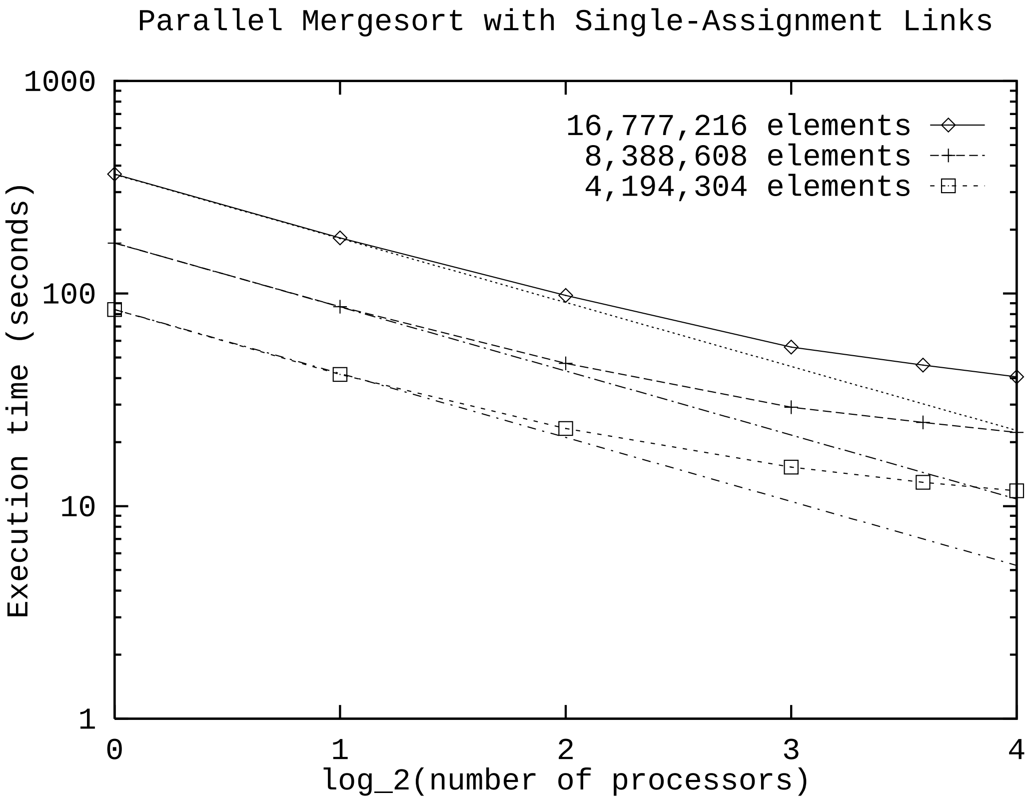


Figure 7.5: Execution time of parallel mergesort for linked lists with single-assignment links compared to ideal speedup of sequential mergesort for linked lists with mutable links (block length = 65,536 elements).

mutable links. The parallelism is conveniently expressed using parallelizable sequences of statements, and the sophisticated and highly dynamic synchronization strategy is elegantly expressed using single-assignment links between nodes in the linked lists. The Priority pragma provides a simple method of improving load balancing. The algorithm would be considerably more complicated if it were expressed using less-structured constructs, e.g., threads with barrier and lock synchronization.

- **Efficiency**:

  Despite the additional synchronization overheads, the parallel mergesort algorithm for linked lists with single-assignment links delivers significantly better speedup than a traditional parallel mergesort algorithm for linked lists with mutable links. We do not know of a more efficient parallel algorithm for sorting linked lists that could be expressed using a less-structured parallel programming model but not with our model.

- **Automatic Parallelization**:

  The parallel mergesort algorithm for linked lists with single-assignment links could not reasonably be expected to be generated automatically from an efficient sequential mergesort algorithm for linked lists. A high-level understanding of the purpose of the algorithm is required to recognize that operations on the links between nodes can be used as synchronization operations in a parallelized version of the algorithm. In addition, automatic program parallelization is difficult for programs that involve pointers to dynamically-allocated data structures, because of potential aliasing.

- **Development Costs**:

  Reasoning, testing, and debugging in the context of the sequential semantics make development of the parallel mergesort algorithm for linked lists with single-assignment links essentially no more difficult than development of a sequential program. Development would be more error-prone in a less-structured parallel programming model, particularly because of the sophisticated and highly dynamic patterns of thread creation and synchronization.

All aspects of the benefits of a high-level parallel programming model with direct control of parallelism are increased for a problem such as mergesort of a linked list in which highly dynamic thread creation and synchronization are required and in which load balancing is non-trivial.

7.3 LU Factorization

In this section, we describe the development of an efficient parallel program to compute the LU factorization of a nonsingular matrix without pivoting. The program demonstrates the use of single-assignment types in the context of array data structures and provides additional examples of the use of parallelizable for-loop statements with arguments. Although the parallel LU factorization algorithm is developed from an efficient sequential LU factorization algorithm, the parallel modifications could not reasonably be expected to be generated automatically from the sequential algorithm. We compare the measured performance of LU factorization using single-assignment types to the measured performance of LU factorization using only parallelizable for-loop statements or barriers for synchronization.

7.3.1 Program Specification

The specification of the parallel LU factorization program is given in Program 7.6. The

```
type Matrix is array (Integer range <>, Integer range <>) of Float;

procedure LU_Factorize (A           : in     Matrix;
                        LU          :    out Matrix;
                        Num_Threads : in     Positive);
-- | requires
-- |     A'Length > 0 and Square(A) and Nonsingular(A) and
-- |     Same_Bounds(A, LU) and Num_Threads ≤ A'Length.
-- | ensures
-- |     Unit_Lower_Triangle(LU)*Upper_Triangle(LU) = A.
```

Program 7.6: Specification of parallel LU factorization.

LU_Factorize procedure computes the unit lower triangular and upper triangular factors of the input matrix, A, and overlays the computed factors in the output matrix, LU, as described in Chapter 1. The components of the matrices are mutable floating-point variables. In the interest of brevity, the specification ignores the imprecision of floating-point arithmetic and the possibility of division by zero due to the absence of pivoting. Num_Threads specifies the number of parallel threads to be used in the algorithm. The algorithms that we present could overwrite the input matrix with the output matrix instead of having separate matrices.

7.3.2 Parallel LU Factorization using Barriers

Before we present a parallel LU factorization algorithm using single-assignment types, we investigate a parallel LU factorization algorithm using only parallelizable for-loop statements or barriers for synchronization. The version of the algorithm using parallelizable for-loop statements is presented in Program 7.7. The algorithm computes the components LU in

```
procedure LU_Factorize (A           : in     Matrix;
                        LU          :    out Matrix;
                        Num_Threads : in     Positive) is
begin
    for I in LU'Range loop
        pragma Parallelizable_Loop(Num_Threads, Pattern => Block);
        for J in I .. LU'Last loop
            declare
                Sum : Float := 0.0;
            begin
                for K in LU'First .. I − 1 loop
                    Sum := Sum + LU(I, K)*LU(K, J);
                end loop;
                LU(I, J) := A(I, J) − Sum;
            end;
        end loop;
        pragma Parallelizable_Loop(Num_Threads, Pattern => Block);
        for J in I + 1 .. LU'Last loop
            declare
                Sum : Float := 0.0;
            begin
                for K in LU'First .. I − 1 loop
                    Sum := Sum + LU(J, K)*LU(K, I);
                end loop;
                LU(J, I) := (A(J, I) − Sum)/LU(I, I);
            end;
        end loop;
    end loop;
end LU_Factorize;
```

Program 7.7: Parallel LU factorization using parallelizable for-loop statements as the only form of synchronization.

a pattern of alternating rows and columns. Within each row and column, the components are computed in parallel using a parallelizable for-loop statement with iterations assigned to parallel threads using the Block pattern. An example of the pattern of computation for parallel LU factorization using only parallelizable for-loop statements or barriers for synchronization with four parallel threads is shown in Figure 7.6.

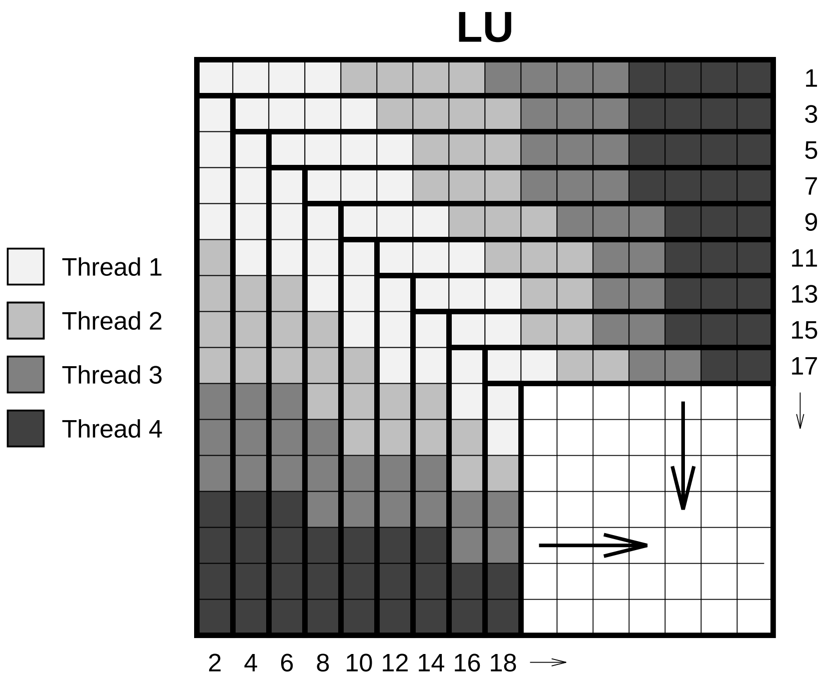


Figure 7.6: An example of the pattern of computation for parallel LU factorization using only parallelizable for-loop statements or barriers for synchronization with four parallel threads.

The same algorithm can be implemented more efficiently by creating one set of long-lived parallel threads and using barrier synchronization between the computation of the rows and columns. In most systems, barrier synchronization is significantly less expensive than thread creation and termination. The version of the algorithm using explicit barrier synchronization is presented in Program 7.8. The complete text of the program is given in Appendix B.4.2. The parallel for-loop statement in this program is a more general construct than our parallelizable for-loop statement. It is not required to be equivalent to a sequential for-loop statement. The LU factorization program using barrier synchronization implements exactly the same pattern of computation as the LU factorization program using only parallelizable for-loop statements for synchronization.

7.3.3 Performance Measurements using Barriers

Figure 7.7 gives the measured speedup of the parallel LU factorization program using barrier synchronization over an efficient sequential LU factorization program. The measurements are for factoring matrices of size 500 by 500 to 1,250 by 1,250 on 1 to 32 processors. In each case, the number of parallel threads is equal to the number of processors, as this was found to give the best speedup. The complete text of the sequential LU factorization program that is used for speedup comparison is given in Appendix B.4.1.

The measurements show that the parallel LU factorization program using barrier synchronization delivers good speedup for small number of processors and that speedup increases with increasing data size, but that speedup peaks at between 8 and 16 processors and declines for more processors. The reason for the declining speedup is that the granularity of parallelism becomes too fine as the number of threads is increased with the number of processors. The overheads of increased synchronization become greater than the performance gained through increased parallel execution, and the load imbalance from threads waiting at barriers becomes more significant as the granularity of parallelism decreases.

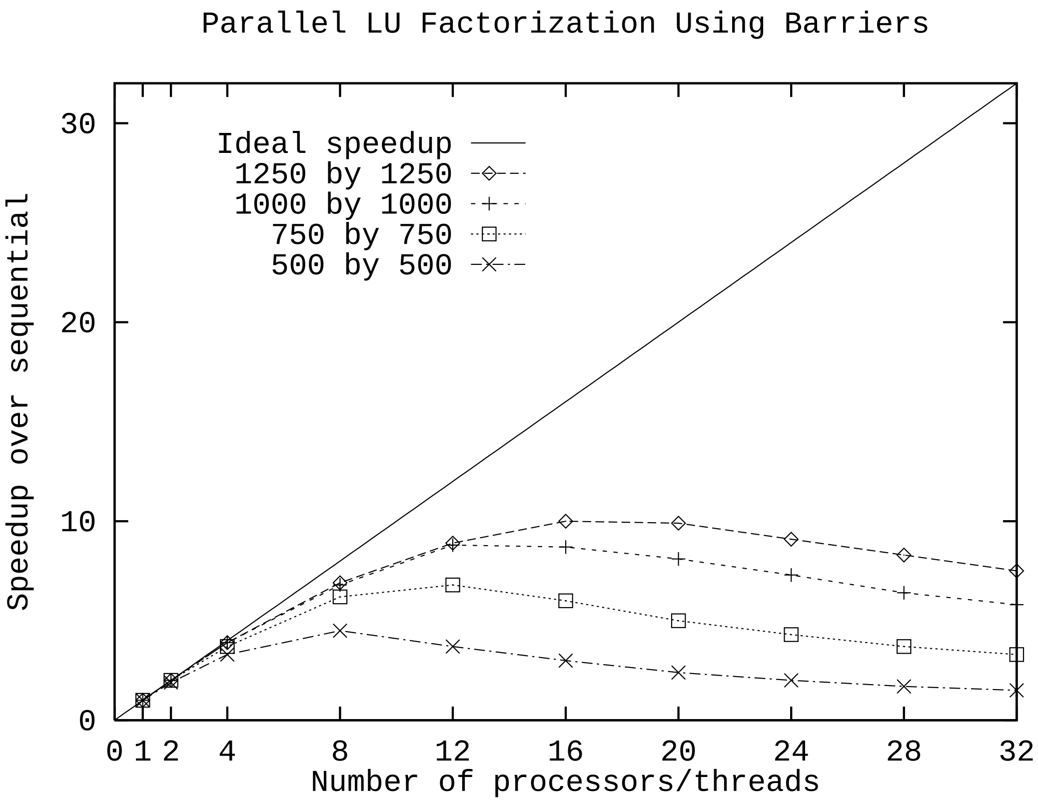
For larger sized matrices, parallel execution on one processor is marginally faster than sequential execution. This behavior is due to caching effects caused by the difference in the order of computation of the components of LU between the parallel and sequential algorithms.

7.3.4 Single-Assignment Flags

A more efficient parallel LU factorization algorithm requires a more sophisticated pattern of synchronization. For this purpose, the definition of the single-assignment flag type is

```
procedure LU_Factorize (A           : in     Matrix;
                        LU          :    out Matrix;
                        Num_Threads : in     Positive) is

    function Split (First, Last, T : Integer) return Integer is
    begin
        return First + Integer((Float(T)/Float(Num_Threads))*Float(Last − First + 1));
    end Split;

    B : Barrier (Num_Threads);

begin
    for T in 0 .. Num_Threads − 1 parallel loop
        for I in LU'Range loop
            for J in Split(I, LU'Last, T) .. Split(I, LU'Last, T + 1) − 1 loop
                declare
                    Sum : Float := 0.0;
                begin
                    for K in LU'First .. I − 1 loop
                        Sum := Sum + LU(I, K)*LU(K, J);
                    end loop;
                    LU(I, J) := A(I, J) − Sum;
                end;
            end loop;
            At_Barrier(B);
            for J in Split(I + 1, LU'Last, T) .. Split(I + 1, LU'Last, T + 1) − 1 loop
                declare
                    Sum : Float := 0.0;
                begin
                    for K in LU'First .. I − 1 loop
                        Sum := Sum + LU(J, K)*LU(K, I);
                    end loop;
                    LU(J, I) := (A(J, I) − Sum)/LU(I, I);
                end;
            end loop;
            At_Barrier(B);
        end loop;
    end loop;
end LU_Factorize;
```

Program 7.8: LU factorization using threads and barriers.



Number of processors	500 by 500		750 by 750		1000 by 1000		1250 by 1250	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
sequential	41.30	—	150.57	—	369.23	—	685.71	—
1	41.33	1.0	152.14	1.0	359.82	1.0	661.48	1.0
2	21.87	1.9	77.06	2.0	183.58	2.0	346.88	2.0
4	12.52	3.3	40.63	3.7	95.15	3.9	177.21	3.9
8	9.28	4.5	24.13	6.2	54.56	6.8	99.08	6.9
12	11.19	3.7	22.16	6.8	41.85	8.8	76.82	8.9
16	13.99	3.0	25.26	6.0	42.51	8.7	68.58	10.0
20	17.57	2.4	30.29	5.0	45.45	8.1	69.33	9.9
24	20.83	2.0	34.79	4.3	50.41	7.3	75.07	9.1
28	23.92	1.7	40.20	3.7	57.89	6.4	82.40	8.3
32	26.66	1.5	46.23	3.3	63.97	5.8	90.95	7.5

All times are in seconds.

Figure 7.7: Speedup of parallel LU factorization using barrier synchronization over sequential LU factorization.

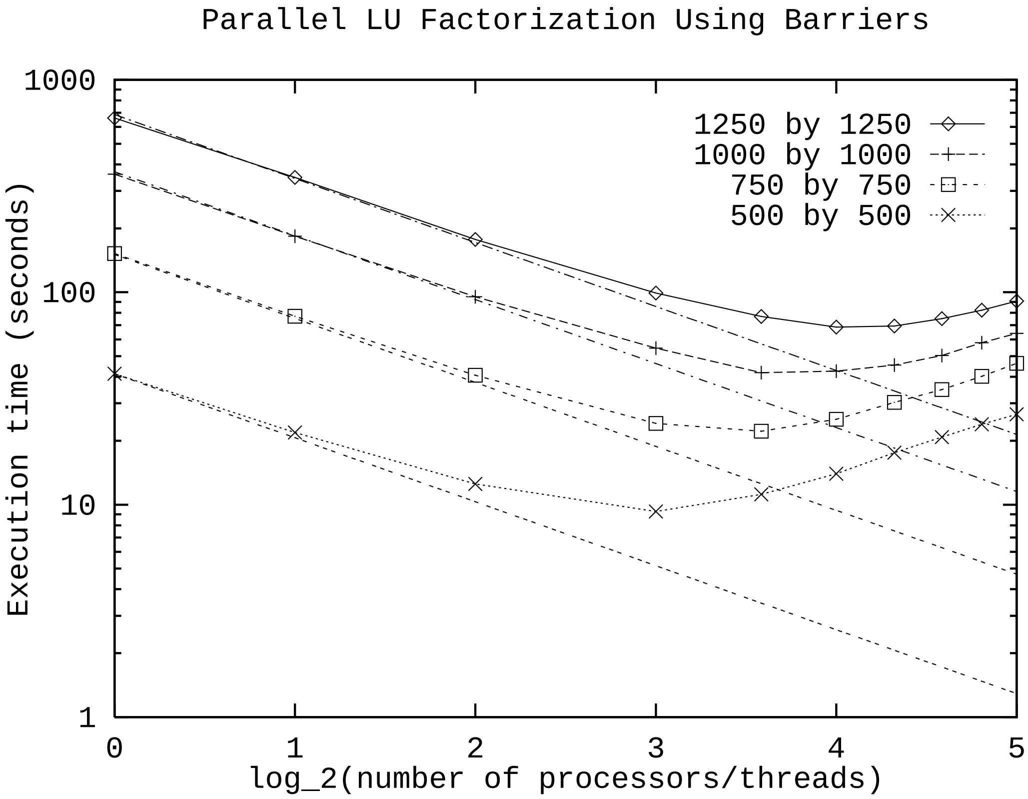


Figure 7.8: Execution time of parallel LU factorization using barrier synchronization compared to ideal speedup of sequential LU factorization.

presented in Program 7.9. A flag is a single-assignment unary variable that can only be

```
type Flag is (Set);
pragma Single_Assignment(Flag);

procedure Set (F : out Flag) is
begin
    F := Set;
end Set;

procedure Check (F : in Flag) is
begin
    if F = Set then null; end if;
end Check;
```

Program 7.9: Definition of the single-assignment flag type.

assigned one possible value. The Set operation assigns to the flag and the Check operation evaluates the flag. A program is erroneous if its sequential interpretation checks a flag that has not been set (violating restriction E.5) or sets the same flag more than once (violating restriction E.6). Therefore, in sequential execution of a non-erroneous program, a Check operation has no effect. In parallel execution of a non-erroneous program, a Check operation suspends until the Set operation is performed on the flag. In this manner, Set and Check operations on single-assignment flags can be used to synchronize parallel threads.

7.3.5 Parallel LU Factorization using Single-Assignment Flags

A more efficient parallel LU factorization algorithm using single-assignment flags for synchronization is presented in Program 7.10. The complete text of the program is given in Appendix B.4.3. Parallel LU factorization using single-assignment flags operates as follows:

1. The components of LU are subdivided into equal-sized square blocks, with each block guarded by a single-assignment flag. An example of the subdivision of LU into blocks is shown in Figure 7.9. In the program, R(B) is the row of block B and C(B) is the column of block B.

2. All the blocks are computed in parallel. The number of blocks and the number of parallel threads are specified separately.

3. At the start of the computation of a block, the flags of the first block above the block in the upper triangle and of the first block to the left of the block in the lower triangle

```
procedure LU_Factorize (A           : in     Matrix;
                        LU          :    out Matrix;
                        Num_Blocks  : in     Positive;
                        Num_Threads : in     Positive ) is

    function Start (I : Integer) return Integer is
    begin
        return LU'First + Integer((Float(I)/Float(Num_Blocks))*Float(LU'Length));
    end Start;

    R, C  : array (0 .. Num_Blocks*Num_Blocks − 1) of Integer;
    Done : array (−1 .. Num_Blocks − 1, −1 .. Num_Blocks − 1) of Flag;

begin
    R := . . . ; C := . . . ;
    for I in 0 .. Num_Blocks − 1 loop
        Set(Done(−1, I)); Set(Done(I, −1));
    end loop;
    pragma Parallelizable_Loop(Num_Threads, Pattern => On_Demand);
    for B in 0 .. Num_Blocks*Num_Blocks − 1 loop
        Check(Done(R(B), Min(R(B), C(B) − 1)));
        Check(Done(Min(C(B), R(B) − 1), C(B)));
        for I in Start(R(B)) .. Start(R(B) + 1) − 1 Loop
            for J in Start(C(B)) .. Min(Start(C(B) + 1), I) − 1 loop
                declare
                    Sum : Float := 0.0;
                begin
                    for K in LU'First .. J − 1 loop
                        Sum := Sum + LU(I, K)*LU(K, J);
                    end loop;
                    LU(I, J) := (A(I, J) − Sum)/LU(J, J);
                end;
            end loop;
            for J in Max(Start(C(B)), I) .. Start(C(B) + 1) − 1 loop
                declare
                    Sum : Float := 0.0;
                begin
                    for K in LU'First .. I − 1 loop
                        Sum := Sum + LU(I, K)*LU(K, J);
                    end loop;
                    LU(I, J) := A(I, J) − Sum;
                end;
            end loop;
        end loop;
        Set(Done(R(B), C(B)));
    end loop;
end LU_Factorize;
```

Program 7.10: LU factorization using single-assignment flags.

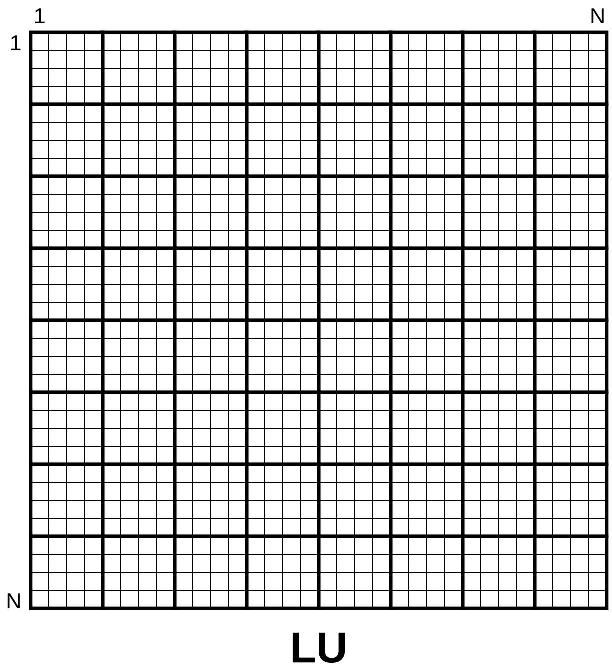


Figure 7.9: An example of the subdivision of LU into blocks for parallel LU factorization using single-assignment flags.

are checked. This implicitly suspends the computation of the block until the blocks on which it depends have been computed. An example of the data dependencies between blocks is shown in Figure 7.10(a).

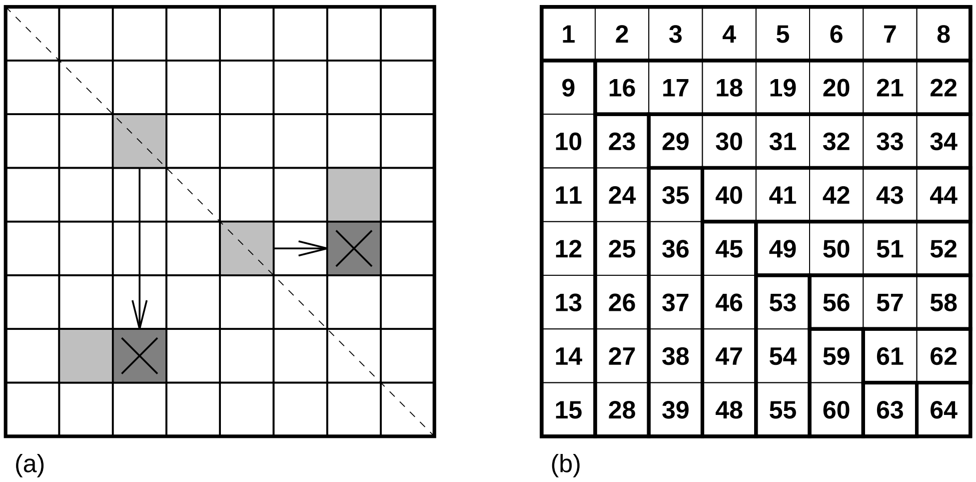


Figure 7.10: Parallel LU factorization using single-assignment flags: (a) An example of the data dependencies between blocks. (b) An example of the order in which blocks are assigned to parallel threads.

4. At the end of the computation of a block, the flag of the block is set. This implicitly awakens the computation of blocks that depend on this block.

5. Apart from checking and setting flags, the computation of a block uses the standard sequential LU factorization algorithm.

6. Blocks are assigned to parallel threads in a pattern of alternating rows and columns using the On_Demand pattern. This pattern is chosen to maximize the number of block computations that are executable at any given time. An example of the order in which blocks are assigned to parallel threads is shown in Figure 7.10(b).

In parallel execution of the program, operations on flags implicitly suspend the computations of blocks until the blocks on which they depend have been computed. In sequential execution of the program, blocks are computed in an order such that flags are always set before they are checked. Therefore, deadlock cannot occur in parallel execution of the program. Since checking flags is the first step and setting the flag is the last step in the computation of a block, it is straightforward to show that the operations on flags synchronize the operations on the mutable components of LU so that restriction E.1 is not violated.

7.3.6 Performance Measurements using Single-Assignment Flags

Figure 7.11 gives the measured speedup of the parallel LU factorization program using single-assignment flags over an efficient sequential LU factorization program. The measurements are for factoring matrices of size 500 by 500 to 1,250 by 1,250 on 1 to 32 processors. In each case, the block size is 10 by 10 components and the number of parallel threads is equal to the number of processors, as this was found to give the best speedup. The complete text of the sequential LU factorization program that is used for speedup comparison is given in Appendix B.4.1.

The measurements show that the parallel LU factorization program using single-assignment flags delivers considerably better speedup than the parallel LU factorization program using barriers for synchronization. For LU factorization of a 1,250 by 1,250 component array, the best time using single-assignment flags is 125 percent faster than the best time using barriers. Since the granularity of parallelism can be varied independently of the number of parallel threads, the speedup scales well then levels off with increasing numbers of processors instead of peaking then declining. The additional performance is obtained without adding much complexity to the program.

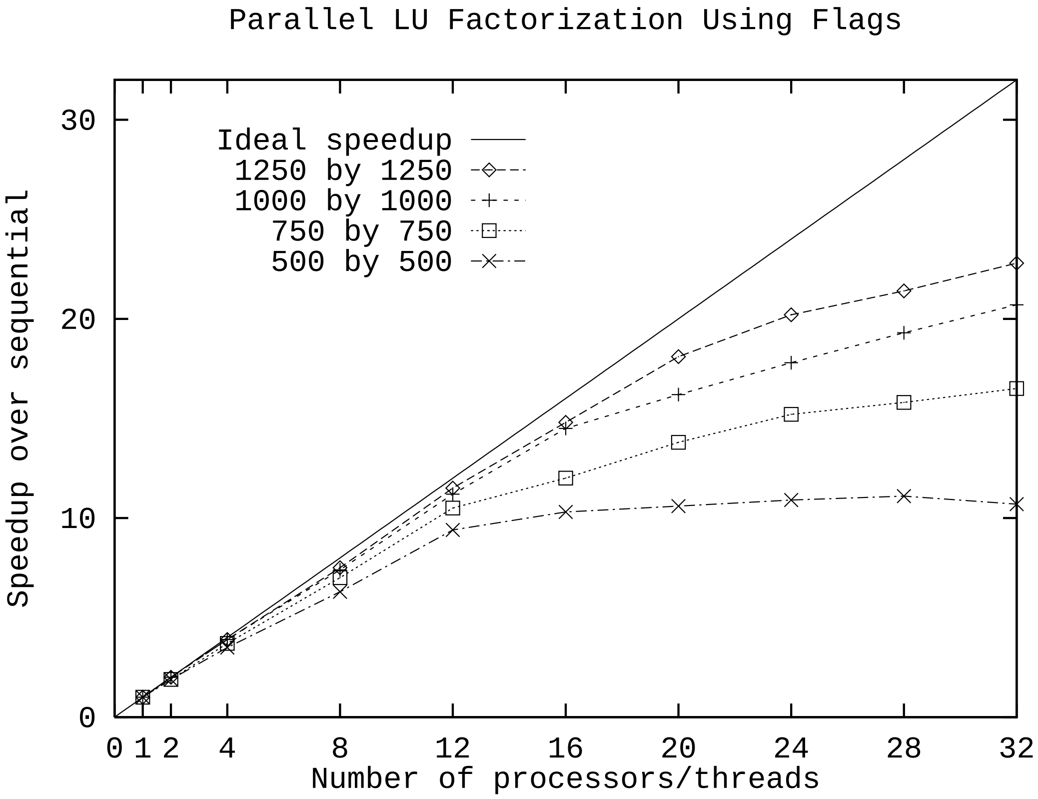
For larger sized matrices, parallel execution on one processor is marginally faster than sequential execution. This behavior is due to caching effects caused by the difference in the order of computation of the components of LU between the parallel and sequential algorithms.

7.3.7 Experimental Analysis

The development of the parallel LU factorization program using single-assignment flags for synchronization supports our experimental propositions as follows:

- **Expressiveness**:

  The parallel LU factorization algorithm using single-assignment flags is almost the same as an efficient sequential LU factorization algorithm. The parallelism and control of granularity are concisely expressed using a parallelizable for-loop statement with iterations assigned to parallel threads using the On_Demand pattern, and the sophisticated synchronization strategy is elegantly expressed using single-assignment flags. The algorithm would be considerably more complicated if it were expressed using less-structured parallel programming constructs, e.g., threads with barrier and lock synchronization.



Number of processors	500 by 500		750 by 750		1000 by 1000		1250 by 1250	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
sequential	41.30	—	150.57	—	369.23	—	685.71	—
1	42.31	1.0	151.35	1.0	356.20	1.0	653.85	1.0
2	22.16	1.9	78.32	1.9	184.02	2.0	344.80	2.0
4	11.81	3.5	40.57	3.7	94.85	3.9	177.24	3.9
8	6.54	6.3	21.48	7.0	50.07	7.4	91.82	7.5
12	4.38	9.4	14.36	10.5	33.07	11.2	59.82	11.5
16	4.00	10.3	12.60	12.0	25.42	14.5	46.43	14.8
20	3.88	10.6	10.95	13.8	22.73	16.2	37.81	18.1
24	3.80	10.9	9.93	15.2	20.80	17.8	33.90	20.2
28	3.73	11.1	9.50	15.8	19.17	19.3	32.11	21.4
32	3.86	10.7	9.14	16.5	17.84	20.7	30.13	22.8

All times are in seconds.

Figure 7.11: Speedup of parallel LU factorization using single-assignment flags over sequential LU factorization (block size = 10 by 10 components).

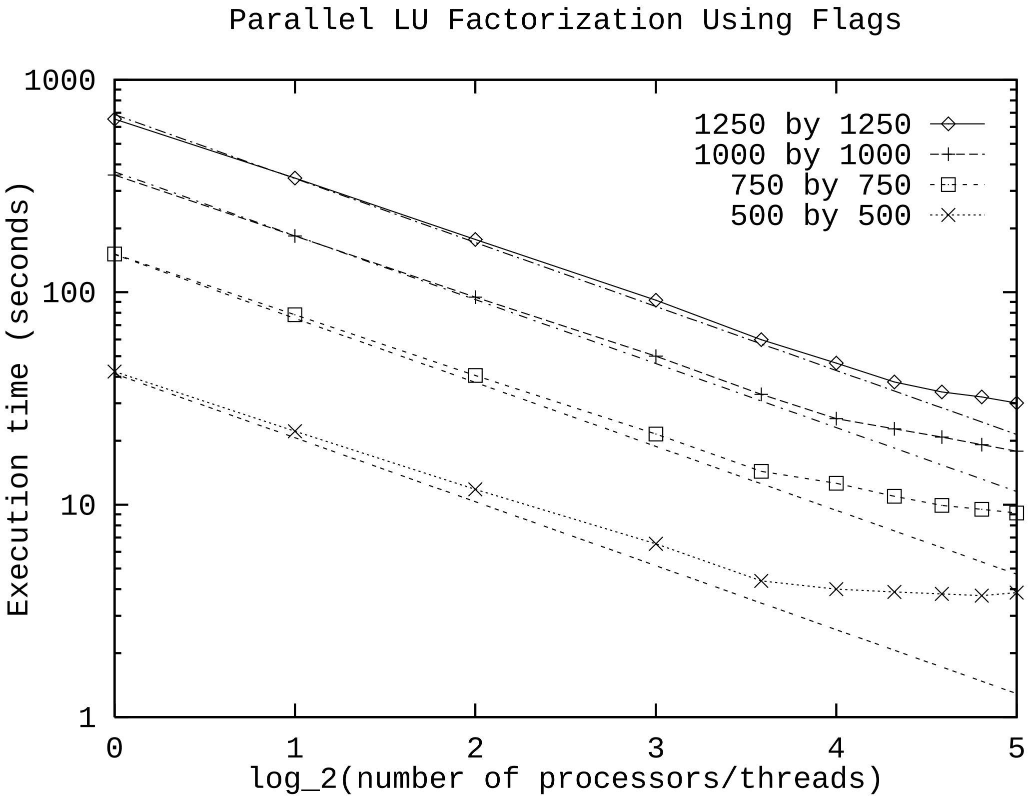


Figure 7.12: Execution time of parallel LU factorization using single-assignment flags compared to ideal speedup of sequential LU factorization.

- **Efficiency**:

  The parallel LU factorization program using single-assignment flags delivers significantly better speedup than the parallel LU factorization program using barriers for synchronization. The algorithm using barriers does not capture all the parallelism that is implicit in the data dependencies. We do not know of a more efficient parallel algorithm for LU factorization without pivoting that could be expressed using a less-structured parallel programming model.

- **Automatic Parallelization**:

  The parallel LU factorization program using barriers for synchronization could reasonably be expected to be generated automatically from an efficient sequential LU factorization program. However, the more efficient parallel LU factorization program using single-assignment flags could not reasonably be expected to be generated automatically. A high-level understanding of the purpose of the program is required to invent the synchronization strategy that allows the computation to be subdivided into blocks and to determine an efficient order for assigning blocks to parallel threads. Entirely new data structures and algorithms are introduced into the program.

- **Development Costs**:

  Reasoning, testing, and debugging in the context of the sequential semantics make development of the parallel LU factorization algorithm using single-assignment flags essentially no more difficult than development of a sequential program. Development would be more error-prone in a less-structured parallel programming model, particularly because of the sophisticated strategies for synchronization and execution scheduling.

The parallel LU factorization program using single-assignment flags is the most sophisticated of all our experimental programs and provides the best demonstration of the benefits of direct control of parallelism in a programming model with sequential semantics.

Chapter 8

Limitations on Nondeterminacy

In this chapter, we investigate the consequences of the limitations on nondeterminacy in our parallel programming model that result from restricting the model so that the parallel and sequential semantics are equivalent. The model cannot express parallel algorithms in which the actions of the computation are affected by the timing of the execution of the parallel threads. We describe an important class of parallel search algorithms that require this form of nondeterminacy and hence cannot be expressed using our model. This leads us to outline how our parallel programming model could be integrated with other models that provide less-restrictive synchronization constructs.

8.1 Limitations on Nondeterminacy in Our Model

In our parallel programming model, communication and synchronization between parallel threads are subject to a set of restrictions designed such that execution according to the parallel semantics is equivalent to execution according to the standard sequential semantics. The benefit of these restrictions is that reasoning, testing, and debugging of a parallel program can be performed in the context of the sequential semantics. The cost of these restrictions is that an important class of efficient parallel algorithms with nondeterministic behavior cannot be expressed using our programming model.

Our parallel programming model cannot express an algorithm in which the actions that are performed by the computation are affected by the timing of the execution of the parallel threads. The interleaving of actions between the synchronization operations of two parallel threads can be affected by the timing of the threads (since all such interleavings are equivalent to sequential execution), but the order of the synchronization operations cannot be affected by the timing of the threads. Essentially, we are restricted to expressing algorithms that are no less deterministic in the actions they perform than sequential algorithms.

For some problems that involve irregular data structures or computational patterns, the most efficient known parallel algorithms make nondeterministic choices affected by timing of the parallel threads. Many examples are provided by parallel branch-and-bound search algorithms to solve combinatorial optimization and operations research problems. For these algorithms, our parallel programming model needs to be integrated with a model that provides less-restrictive synchronization constructs, e.g., locks, semaphores, or monitors.

8.2 A Simple Example

Before we consider more complicated examples, we present a simple example that demonstrates the limitations on nondeterminacy in our parallel programming model. A parallel algorithm to sum the components of a two-dimensional array of integers using single-assignment types for synchronization is presented in Program 8.1. The rows of the array are independently summed in parallel with each other, with the row sums assigned to an array of single-assignment integers. The array of row sums is summed in parallel with the summation of the rows. The summation of row sums implicitly suspends whenever an unassigned row sum is evaluated and resumes when the row sum is assigned.

With this algorithm, regardless of the timing of the row summations, the row sums are always summed in the same order. In the worst case (when the first row sum is the last to be assigned), all N steps in the summation of the row sums remain to be executed after the summation of the rows completes. On average, N/2 steps in the summation of the row sums remain to be executed after the summation of the rows completes. This could be improved to $\log_2 N$ steps by building a binary tree of partial sums, at the cost of $N-1$ extra assignments and evaluations of single-assignment variables. A more efficient algorithm is to sum the row sums in the order in which the row summations complete, but this algorithm cannot be expressed using our programming model.

An efficient parallel algorithm to sum the components of a two-dimensional array using a less-restrictive programming model with locks for synchronization is presented in Program 8.2. The rows of the array are independently summed in parallel with each other and each row sum is immediately added to the total sum. A lock is used to ensure mutual exclusion in the updating of the total sum. The parallel for-loop statement in this program is not required to be equivalent to a sequential for-loop statement.

With this algorithm, the order in which the row sums are summed is nondeterministically controlled by the order in which the row summations complete and acquire the mutual

```
type Matrix is array (1 .. N, 1 .. N) of Integer;

function Sum_Of_Components (A : Matrix) return Integer is

    type Single_Integer is new Integer;
    pragma Single_Assignment(Single_Integer);

    Row_Sum : array (1 .. N) of Single_Integer;
    Sum     : Integer := 0;

begin
    begin
        pragma Parallelizable_Sequence;
        pragma Parallelizable_Loop;
        for I in 1 .. N loop
            declare
                Local_Sum : Integer := 0;
            begin
                for J in 1 .. N loop
                    Local_Sum := Local_Sum + A(I, J);
                end loop;
                Row_Sum(I) := Single_Integer(Local_Sum);
            end;
        end loop;
        for I in 1 .. N loop
            Sum := Sum + Integer(Row_Sum(I));
        end loop;
    end;
    return Sum;
end Sum_Of_Components;
```

Program 8.1: Parallel summation of the components of a two-dimensional array using single-assignment types for synchronization.

```
type Matrix is array (1 .. N, 1 .. N) of Integer;

function Sum_Of_Components (A : Matrix) return Integer is

    Sum : Integer := 0;
    L   : Lock;

begin
    for I in 1 .. N parallel loop
        declare
            Row_Sum : Integer := 0;
        begin
            for J in 1 .. N loop
                Row_Sum := Row_Sum + A(I, J);
            end loop;
            Acquire(L);
            Sum := Sum + Row_Sum;
            Release(L);
        end;
    end loop;
    return Sum;
end Sum_Of_Components;
```

Program 8.2: Parallel summation of the components of a two-dimensional array using locks for synchronization.

exclusion lock. A consequence of the nondeterminacy in the order of summation is the possibility of nondeterminacy in the results of the program. Addition of bounded integers is not associative with respect to overflow. Program 8.1 is deterministic with respect to exceptional termination, regardless of whether the program is executed according to the parallel or sequential semantics. However, for some input matrices, Program 8.2 may be nondeterministic with respect to exceptional termination.

8.3 Restrictions on Clocks

The equivalence of the parallel and sequential semantics of our programming model requires restrictions on parallel access to clocks. These restrictions are no more than a special case of the restrictions on parallel access to shared mutable variables. Consider the following program:

```
declare
    X, Y : Time;
begin
    begin
        pragma Parallelizable_Sequence;
        X := Clock;
        Y := Clock;
    end;
    if X <= Y then Put_Line("Yes"); else Put_Line("No"); end if;
end;
```

Clock is a standard function that returns the current time of day. Sequential execution of the program will always print "Yes" as output, whereas parallel execution may nondeterministically print either "Yes" or "No" as output. The reason that parallel and sequential execution are not equivalent is that the program violates restriction E.1 regarding parallel access to shared mutable variables.

We model time as a global mutable variable. Execution of every action in a program nondeterministically increases time by a nonnegative amount. Therefore, it is erroneous for any statement of a parallelizable sequence of statements to evaluate the time, because the other statements of the parallelizable sequence of statements are implicitly assigning to the time. (It is not considered erroneous that the statements of a parallelizable sequence of statements assign to the time in parallel, because this is unobservable.) In other words, it is erroneous to call the Clock function within a parallelizable sequence of statements or a parallelizable for-loop statement.

Note that it is not the nondeterminacy of time in itself that leads to the restrictions.

Rather, it is the difference in the determinacy of time between parallel and sequential execution of a program. Consider the following similar program:

```
declare
    X, Y : Integer;
begin
    begin
        pragma Parallelizable_Sequence;
        X := Random;
        Y := Random;
    end;
    if X <= Y then Put_Line("Yes"); else Put_Line("No"); end if;
end;
```

If Random is a true-random function that returns a different randomly chosen integer on each call, the program is nondeterministic, but is not erroneous. In this case, both sequential and parallel execution will nondeterministically print either "Yes" or "No" as output. If Random is a pseudo-random function that updates a mutable seed variable on each call, the program is erroneous. In this case, for a given initial seed value, sequential execution will always print the same output value, whereas parallel execution may print either "Yes" or "No" as output.

8.4 A Class of Nondeterministic Parallel Algorithms

In this section, we describe parallel branch-and-bound algorithms as a practical example of the limitations on nondeterminacy in our parallel programming model. Branch-and-bound [79][92] is a problem solving strategy that forms the basis of search algorithms to solve a large number of important problems in combinatorial optimization and operations research. Branch-and-bound algorithms are typically used for NP-hard problems in which the search space is exponentially large with respect to the size of the problem. Examples of problems for which branch-and-bound algorithms provide efficient solutions include the 0-1 Knapsack problem [62], the Traveling Salesman problem [84], and Integer Programming [77].

The size and shape of the search space that is traversed by a branch-and-bound algorithm cannot be predicted in advance. Therefore, in a parallel branch-and-bound algorithm, it is not possible to efficiently partition the search space among the parallel threads in any predetermined manner. Parallel branch-and-bound algorithms in which the parallel threads asynchronously respond to partial search results from other threads are more efficient than algorithms in which the interactions between the parallel threads are synchronous. We compare a synchronous and deterministic parallel branch-and-bound algorithm expressed using our parallel programming model with an asynchronous and nondeterministic parallel branch-and-bound algorithm expressed using a less-restrictive model.

8.4.1 Sequential Branch-and-Bound

Suppose we wish to find the solution from a constrained solution space that maximizes a given objective function. Let S be the best feasible solution identified thus far, let Z be the objective function value for S, and let Q be a queue of subsets of the solution space, maintained in descending order of upper bounds on their objective function values. The basic branch-and-bound strategy for finding an exact optimal solution using best-first search operates as follows:

Initially: S is undefined, Z is negative infinity, and Q contains a single subset representing the entire solution space.

Branching Step: Remove the subset from the head of Q and partition it into a collection of more tightly constrained subsets of the solution space.

Bounding Step: For each of the subsets generated in the branching step:

1. If the subset is known to contain no feasible solutions, discard the subset.

2. If the subset is known to contain a single feasible solution, compute the objective function value for the feasible solution. If the objective function value is greater than Z, replace S with the feasible solution, replace Z with the objective function value, and delete any subsets from Q with upper bounds less than or equal to the new value of Z. Otherwise, discard the subset.

3. If the subset may contain more than one feasible solution, compute an upper bound on the objective function value for the subset. If the upper bound is greater than Z, insert the subset into Q according to its upper bound. Otherwise, discard the subset.

Termination Test: If Q is empty, S is an optimal feasible solution and Z is the objective function value for the optimal feasible solution. Otherwise, return to the branching step.

To minimize rather than maximize the objective function, lower bounds are computed instead of upper bounds and the queue is maintained in increasing order of lower bounds instead of decreasing order of upper bounds. Variations on the basic branch-and-bound strategy are possible, including different methods of choosing the subset to branch on at each iteration and variations that find a solution within a specified percentage of the optimal objective function value instead of an exact solution. An example of one iteration of the basic branch-and-bound strategy is shown in Figure 8.1.

8.4.2 An Example: The 0-1 Knapsack Problem

The objective in the 0-1 Knapsack problem is to load a knapsack with a set of indivisible objects, such that the total value of the included objects is maximized and the capacity of the knapsack is not exceeded. The problem can be formulated as follows:

$$\begin{aligned} \text{Maximize} \quad & \sum_{i=1}^{n} v_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq C \\ & x_i \in \{0,1\},\ 1 \leq i \leq n \end{aligned}$$

where C is the capacity of the knapsack, n is the number of objects, and object i has value v_i and weight w_i. The 0-1 Knapsack problem is NP-complete, and therefore the most efficient known algorithms have running time that is exponential with respect to n.

An algorithm to solve the 0-1 Knapsack problem can be developed from the branch-and-bound framework as follows:

1. A subset of the solution space consists of an assignment of *included*, *excluded*, or *free* to each of the n objects. A subset is feasible if the total weight of the included objects is less than or equal to the capacity of the knapsack, and is infeasible if the total weight of the included objects is greater than the capacity of the knapsack. A feasible subset represents a single feasible solution if none of the objects are free.

2. Branching is performed by choosing one of the free objects and generating two new subsets of the solution space: one with the chosen object included, and the other with the chosen object excluded. The strategy for choosing the free object to branch on may affect the performance of the algorithm, but does not affect the correctness of the algorithm.

3. An upper bound on a feasible subset is computed by relaxing the indivisibility of the objects and loading the remaining capacity of the knapsack with the free objects in order of increasing "value density" (v_i/w_i).

The efficiency of the algorithm may be improved by presorting the objects into decreasing order of value density and branching on objects in this order. This simple branch-and-bound algorithm is discussed in more detail by McKeown et al. [89].

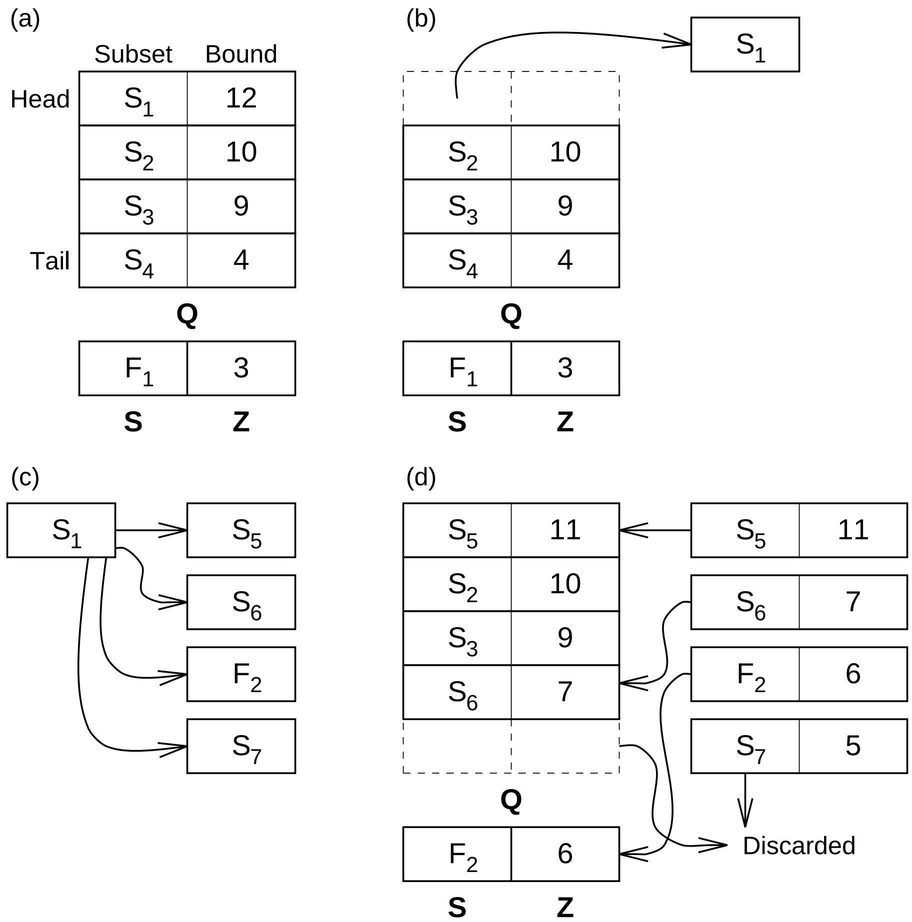


Figure 8.1: An example of one iteration of the basic branch-and-bound strategy: (a) The queue of subsets. (b) The subset with the highest upper bound is removed from the head of the queue. (c) The subset is partitioned into a collection of subsets. (d) The subsets are inserted into the queue according to their upper bounds or discarded.

8.4.3 Synchronous Parallel Branch-and-Bound Using Our Model

A synchronous and deterministic parallel branch-and-bound strategy can be expressed using our parallel programming model. The parallel strategy is a straightforward adaptation of the sequential strategy, as follows:

- On each iteration, the first P subsets of the solution space are removed from Q, instead of just one subset. Each of these subsets is assigned to a separate parallel thread.
- Each parallel thread performs the standard branching and bounding steps on its assigned subset, but stores the resulting subsets in a local queue, instead of inserting them directly into Q.
- After all the parallel threads have completed their branching and bounding steps, the resulting subsets are merged into Q.

This parallel strategy can be implemented using a parallelizable for-loop statement inside the main loop without additional synchronization, or using long-lived threads with barrier synchronization between iterations. This strategy will always visit exactly the same solutions in its search and will always return the same optimal solution, regardless of whether execution is parallel or sequential.

The synchronous parallel branch-and-bound strategy suffers from a number of shortcomings with regard to efficiency:

1. Poor load balancing can result from the synchronization of the parallel threads. This problem is more severe if the branching and bounding time is highly variable.
2. The sequential component of each iteration limits the achievable speedup. This problem is more severe for larger numbers of processors.
3. Inefficient search ordering and unnecessary searching can result from the inability of one parallel thread to asynchronously communicate updated search bounds to other parallel threads. This problem is more severe for larger numbers of parallel threads.

Published experimental results [78][89] indicate that completely synchronous parallel branch-and-bound algorithms deliver limited speedups.

8.4.4 Asynchronous Parallel Branch-and-Bound Using Locks

An asynchronous and nondeterministic parallel branch-and-bound strategy can be expressed in a less-restrictive parallel programming model that provides locks for synchronization.

Again, the parallel strategy is a straightforward adaptation of the sequential strategy, as follows:

- Long-lived parallel threads are created during the initialization phase of the algorithm.
- Each thread repeatedly removes the subset from the head of Q, performs the standard branching and bounding steps, then inserts the resulting subsets directly into Q. There is no synchronization of the parallel threads between iterations.
- Mutual exclusion of operations on Q is obtained through the use of locks.

Since the order of the operations on the shared queue is affected by the timing of the parallel threads, this algorithm cannot be expressed using our parallel programming model. Although the solution returned will always have the same optimal objective function value, in different executions this strategy may visit different solutions in its search and may return different optimal solutions.

The asynchronous parallel branch-and-bound strategy does not suffer from the shortcomings of the synchronous strategy with respect to efficiency:

1. Load balancing is improved, since parallel threads are not required to synchronize at the end of each iteration.
2. There is no sequential component to limit speedup.
3. The efficiency of search ordering is improved, since each parallel thread is always able to choose the currently best subset to search.

Many modifications to this strategy are possible to reduce the performance hot spot of the shared queue and to asynchronously communicate updated bounds between parallel threads without creating a communication bottleneck. A comprehensive discussion of the alternatives is given by McKeown et al. [89]. Published experimental results [78][85][89][99][137] indicate that asynchronous parallel branch-and-bound algorithms can deliver good speedups for a wide range of problems and multiprocessor architectures.

8.5 Integration of Our Model with Less-Restrictive Models

There are two reasons that we might want to integrate our parallel programming model with some other model that provides less-restrictive synchronization constructs:

1. To write programs to solve explicitly concurrent problems that do not have sequential solutions, e.g., real-time monitoring and control problems.

2. To express parallel algorithms in which the actions of the computation are affected by the timing of the parallel threads, e.g., asynchronous parallel branch-and-bound algorithms.

In both cases, we may want to write part of the program using a less-restrictive model in which parallel execution is not equivalent to sequential execution, and part of the program using our parallel programming model.

It is relatively straightforward to define a framework that allows a parallel program written using our model to be embedded in a parallel program written using another model. Provided there are no communication and synchronization operations between the embedded program and the enclosing program, the embedded program is equivalent to a sequential program. Similarly, a parallel program written using another model can be embedded in a parallel program written using our model, provided there are no communication and synchronization operations between the embedded program and the enclosing program. As an example, we outline the integration of our parallel programming model with the standard Ada tasking model.

8.5.1 The Ada Tasking Model

The Ada tasking model [3, section 9] is a shared-memory parallel programming model with a powerful set of structured constructs for task creation and termination, communication and synchronization between tasks, and control of task timing and scheduling.

Task Declaration, Creation, and Termination

A task is a parallel thread of control that is created by either: (i) declaration of an object of a task type, or (ii) dynamic allocation of an object of a task type. Task types may be declared at any level of nesting and may include local declarations, in the same manner as subprogram and package declarations. A task type declaration may include arguments to be specified when task objects of that type are created. Task objects may be passed as arguments to subprogram calls but may not be assigned to other task objects. A task terminates when it completes execution or when it is explicitly aborted. Termination of a block implicitly suspends until all task objects declared in the block have terminated and all dynamically-allocated task objects designated by access types declared in the block have terminated.

Program 8.3 presents an example of parallel matrix multiplication using tasking, with every row of the Result matrix computed by a separate dynamically-allocated task.

```
type Matrix is array (0 .. N − 1, 0 .. N − 1) of Float;

procedure Multiply (A, B : in Matrix; Result : out Matrix) is

    task type Row (I : Integer);
    task body Row is
        Sum : Float;
    begin
        for J in 0 .. N − 1 loop
            Sum := 0.0;
            for K in 0 .. N − 1 loop
                Sum := Sum + A(I, K)*B(K, J);
            end loop;
            Result(I, J) := Sum;
        end loop;
    end Row;

    type Row_Access is access Row;
    Compute_Row : Row_Access;

begin
    for I in 0 .. N − 1 loop
        Compute_Row := new Row (I);
    end loop;
end Multiply;
```

Program 8.3: Parallel matrix multiplication using tasking.

Communication and Synchronization

Tasks communicate and synchronize using the following mechanisms:

- Tasks implicitly synchronize with their parent task and other tasks created in the same block during activation and termination.
- Protected objects are shared data objects (similar to a limited form of monitor) that provide mutual exclusion and suspension on entry conditions.
- Rendezvous provides synchronized communication when one task executes an entry call on a declared entry of another task and the other task executes an accept on the entry.

- Access to shared variables is permitted, subject to synchronization restrictions that are equivalent to the restrictions on shared mutable variables in our model.

Select statements provide a means of waiting for any of a number of specified alternative communication and synchronization events, which may include one or more conditional accept statements, termination if all dependent tasks have completed, or a timed delay. Task priorities may be specified to control scheduling.

Program 8.4 presents an example of a producer task and a consumer task communicating via a rendezvous operation. Program 8.5 presents an example of a group of producer tasks and consumer tasks communicating via a bounded buffer implemented as a protected object.

8.5.2 Integration of Our Model with the Ada Tasking Model

The execution of a program that contains our pragmas and no tasking constructs according to the parallel semantics is equivalent to the execution of the program according to the standard semantics of Ada. We would like a set of restrictions on the allowable interactions between our pragmas and tasking constructs that extends the equivalence result to programs that contain tasking constructs. We outline a framework for integration in which:

- a parallel program written using our model can be embedded within a parallel program written using tasking, and
- a parallel program written using tasking can be embedded within a parallel program written using our model.

Arbitrary nesting of programs written in the two models is permitted. We outline a conservative set of restrictions that ensure that: (i) threads created using our model synchronize with each other using only single-assignment variables, (ii) tasks synchronize with each other using only standard tasking synchronization constructs, and (iii) an embedded program synchronizes with the program in which it is embedded only at initiation and termination. With these restrictions, an embedded parallel program is equivalent to a (possibly nondeterministic) embedded sequential program.

Embedding Our Model Within Tasking

The following restrictions ensure that parallel execution of a parallelizable sequence of statements or parallelizable for-loop statement embedded within tasking is equivalent to sequential execution:

```
type Element is ... ;

task Consumer is
    entry Receive (E : in Element);
end Consumer;

task body Consumer is
    Item : Element;
begin
    loop
        select
            accept Receive (E : in Element) do
                Item := E;
            end Receive;
            Consume(Item);
        or
            terminate;
        end select;
    end loop;
end Consumer;

task Producer;

task body Producer is
    Item     : Element;
    Finished : Boolean;
begin
    Finished := False;
    while not Finished loop
        Produce(Item, Finished);
        Consumer.Receive(Item);
    end loop;
end Producer;
```

Program 8.4: A producer task and a consumer task communicating via a rendezvous operation.

```
type Element  is ... ;
type Elements is array (Integer range <>) of Element;

protected type Bounded_Buffer (Length : Integer) is
    entry Put (Item : in Element);
    entry Get (Item : out Element);
private
    Data  : Elements (1 .. Length);
    Head  : Integer := 1;
    Tail  : Integer := 1;
    Count : Integer := 0;
end Bounded_Buffer;

protected body Bounded_Buffer is
    entry Put (Item : in Element) when Count < Length is
    begin
        Data(Tail) := Item;
        Tail := Tail mod Length + 1;
        Count := Count + 1;
    end Put;
    entry Get (Item : out Element) when Count > 0 is
    begin
        Item := Data(Head);
        Head := Head mod Length + 1;
        Count := Count - 1;
    end Get;
end Bounded_Buffer;

Buffer : Bounded_Buffer (Buffer_Length);

task type Producer is
begin
    ... Buffer.Put(Item); ...
end Producer;

task type Consumer is
begin
    ... Buffer.Get(Item); ...
end Consumer;

Producers  : array (1 .. Num_Producers ) of Producer;
Consumers : array (1 .. Num_Consumers) of Consumer;
```

Program 8.5: A group of producer tasks and consumer tasks communicating via a bounded buffer implemented as a protected object.

- It is illegal for a parallelizable sequence of statements/for-loop statement to contain an accept statement or select statement for the enclosing task declaration.
- It is erroneous for a parallelizable sequence of statements/for-loop statement to perform an entry call on a task created at an outer dynamic level.
- It is erroneous for more than one statement/iteration of a parallelizable sequence of statements/for-loop statement to perform an operation on the same protected object.

An example of the usefulness of embedding our model within tasking is as follows: In an asynchronous parallel branch-and-bound algorithm, if the branching and bounding operations can themselves be parallelized, we might write the parallel branch-and-bound framework using tasking and write the parallelized branch and parallelized bound operations using our parallel programming model.

Embedding Tasking Within Our Model

The following restriction ensures that parallel execution of a parallelizable sequence of statements or parallelizable for-loop statement with embedded tasking is equivalent to sequential execution:

- It is erroneous for a task to perform any operation on a single-assignment variable created at an outer dynamic level.

An example of the usefulness of embedding tasking within our model is as follows: In a highly parallel LU factorization algorithm, the inner loop to compute the inner product of a row and column of LU could be parallelized as well as the outer loops. We might parallelize the outer loops using our parallel programming model and write an inner product routine using tasking, so that the partial sums of the inner product can be summed in the order in which they become available instead of in a predetermined order.

Chapter 9

Distributed Memory

In this chapter, we discuss the issues involved in the implementation, programming, and performance of our parallel programming model on distributed-memory computer systems. The shared-memory model on which our parallel programming model is based can be implemented transparently in hardware or software on top of a distributed-memory system. However, the structure of the underlying memory system has a significant impact on program performance and must be considered in the design of efficient parallel algorithms.

9.1 Overview

Our parallel programming model is defined in the context of a shared-memory model in which a single address space is shared by all parallel threads. This shared-memory model is supported directly by symmetric multiprocessors such as the 36-processor SGI Challenge that we use for our performance experiments. A straightforward approach to implementing a shared-memory programming model on top of a distributed-memory system is to implement all remote memory references as communication operations. With this approach, explicit copying of remote data into local memory is often required for efficiency. A more sophisticated approach is to implement, either in hardware or software, a general "distributed shared-memory" system in which copying of data between remote and local memory is automatic and maintains memory coherence.

With distributed-memory computer systems, two important programming and performance issues arise that are not concerns with shared-memory computer systems:

1. Data distribution: the distribution of data across the distributed memory space.

2. Process mapping: the mapping of parallel processes or threads to processors.

These issues are an integral part of the design of an efficient algorithm for a distributed-memory computer system, as described by Van de Velde [132, Chapter 12]. In addition, even with a distributed shared-memory system, parallel performance is often improved by explicit copying of remote data into local memory. Development of an efficient program for any kind of computer system (uniprocessor or multiprocessor) requires careful consideration of all aspects of the interaction of the program with the memory hierarchy.

9.2 Shared Memory

Before we consider distributed-memory computer systems, we briefly outline programming and performance issues for shared-memory computer systems with uniform memory access, as typified by symmetric multiprocessors. A symmetric multiprocessor consists of a group of processors that share uniform access to the memory system via an interconnection network. The processors normally have one or more levels of caching, and the memory may be interleaved in several banks to reduce memory latency and memory contention. A symmetric multiprocessor system with four processors and two memory banks is shown in Figure 9.1. The most common interconnection network consists of one or more buses to

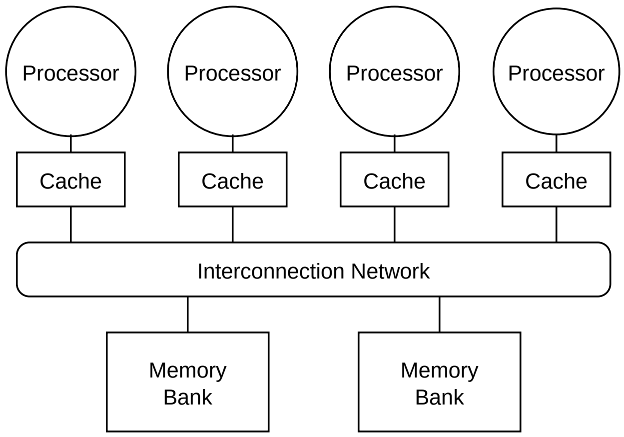


Figure 9.1: A symmetric multiprocessor system.

which all processors and memory banks are connected. Examples of symmetric multiprocessor systems include the Sequent Symmetry, the SGI Challenge and Power Challenge, and the Sun SparcServer 1000 and SparcCenter 2000.

Data distribution is no more complex than for a uniprocessor, because all memory locations are uniformly accessible by all processors using ordinary read and write operations. Memory coherence is maintained automatically by the memory management and interconnection network hardware. Process mapping is not an issue, because threads are dynamically scheduled and mapped onto processors by the operating system from a single ready-queue. Threads can migrate transparently across processors each time they are rescheduled. The operating system may provide options to bind a thread to a subset of the processors, e.g., to ensure a certain responsiveness or fairness, but this provision is normally required only for real-time programs.

The behavior of the caching system is extremely important to the performance of parallel programs executed on a symmetric multiprocessor. Without effective caching, parallel speedups are severely limited by memory contention. Therefore, development of an efficient parallel program for a symmetric multiprocessor requires consideration of the interaction between the pattern of memory accesses and the operation of the caching system. Consideration of cache behavior is also important in the development of efficient programs for uniprocessor computer systems.

The major problem with symmetric multiprocessors is limited scalability. Although caching and memory interleaving can overcome this problem for a moderate number of processors, memory contention eventually becomes a performance bottleneck for a large number of processors. Our own experiments and those of others [47] indicate that bus-based symmetric multiprocessors can scale to at least many tens of processors with current technology, but scalability to many hundreds or thousands of processors is unlikely.

9.3 Distributed Memory

A distributed-memory computer system consists of a group of processing nodes, each with its own local memory, that can exchange messages via an interconnection network. Each processing node may consist of one or more processors. A distributed-memory computer system with four single-processor nodes is shown in Figure 9.1. Possible interconnection networks include crossbar switches, omega networks, hypercubes, and meshes. Examples of distributed-memory computer systems include the Caltech Cosmic Cube, the IBM SP-1 and SP-2, the Intel Delta and Paragon, the SGI Power Challenge Array, and networks of workstations. An excellent review and comparison of shared-memory and distributed-memory computer systems is given by Lenoski and Weber [81, chapter 1].

A significant difference between a distributed-memory computer system and a shared-memory computer system is that a processing node in a distributed-memory computer

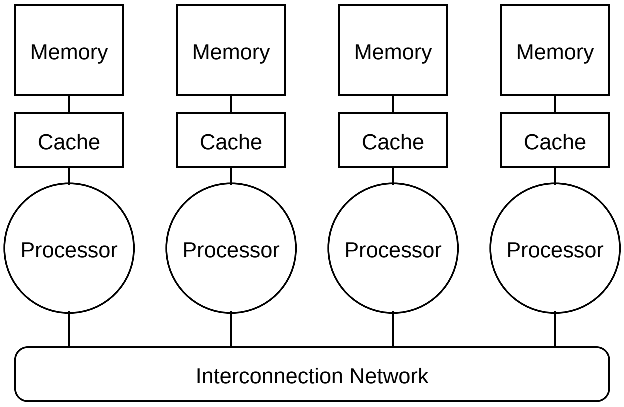


Figure 9.2: A distributed-memory computer system.

system cannot directly access the entire memory space. Each node can directly access its own local memory using ordinary read and write operations and can exchange messages with other nodes, but it cannot directly access the local memories of other nodes. No hardware support is provided for maintaining coherent copies of data across the caches and local memories of multiple nodes.

Data distribution and process mapping are an integral part of the design of efficient algorithms for distributed-memory computer systems. Methods for data distribution include: (i) distribution specifications separate from the program, e.g., Ada 95 [3, Annex E], (ii) annotations to data declarations, e.g., HPF [40][74], and (iii) dynamic memory allocation on different nodes, e.g., PCN [24][42], CC++ [22], Fortran M [45], PVM [119], and MPI [34][115]. Methods for process mapping include: (i) mapping specifications separate from the program, e.g., Ada 95, (ii) annotations and arguments to process creation, e.g., PCN, CC++, PVM and MPI, and (iii) dynamic process migration, e.g., the Concurrent Graph Library [122]. In some systems, data distribution and process mapping can be specified in terms of virtual topologies [41][44][121] that are separately mapped onto actual machine configurations.

A straightforward approach to implementing our parallel programming model on top of a distributed-memory system is to treat the memories as a single address space and implement all remote memory references as message exchanges between processors. Restricting the allowable remote operations, as in CC++ and Ada 95, may simplify implementation and improve efficiency. Since remote memory references will be expensive relative to local memory references, explicit transfer of data between remote and local memories is likely to be required for efficiency in a program. This may add a significant degree of complexity to the program, relative to programming a shared-memory computer system. Essentially, the programmer is required to explicitly cache remote data items in local memory and maintain memory coherence, as in explicit message-passing programming.

9.4 Distributed Shared Memory

A more sophisticated approach to supporting a shared-memory programming model on a distributed-memory computer system is to implement a distributed shared-memory system, i.e., a virtual shared memory with automatic caching of remote data and maintenance of memory coherence. Distributed shared memory is implemented in hardware by machines such as the Stanford DASH and FLASH and the Kendall Square Research KSR-1 and KSR-2. Distributed shared memory is implemented in software by packages such as Ivy [82] and Treadmarks [7] that are designed to run on top of networks of workstations and other distributed-memory computer systems. Most distributed shared-memory systems implement a model with both non-shared local memory and shared global memory.

Distributed shared memory can be implemented in hardware through scalable cache coherence protocols that use distributed cache directories. For increased performance, these protocols generally support weak memory consistency models such as the release consistency model [48] instead of the stronger sequential memory consistency model [76] supported by most bus-based shared-memory multiprocessors. Release consistency is equivalent to sequential consistency for programs that do not violate certain restrictions on access to shared variables between synchronization points. These restrictions are part of most parallel programming models, including our model. A review of approaches to the implementation of distributed shared memory in hardware (and of the design of the Stanford DASH architecture) is given by Lenoski and Weber [81].

Distributed shared memory can be implemented in software through adaptation of paging mechanisms to automatically trap accesses to nonresident remote memory pages and copy those pages to local memory. Coherence of replicated memory pages is maintained using protocols similar to those used to maintain cache coherence in hardware. However, the cost of cache misses and false sharing is greater, because memory pages are generally much larger than cache lines. A review of page-based algorithms for the implementation of distributed shared memory in software is given by Stumm and Zhou [118]. A different approach, based on caching at the level of data objects rather than pages and using single-assignment values for synchronization, is used in the SAM system [104].

Our parallel programming model could be implemented on top of any system that implements distributed shared memory in hardware or software. Such an implementation would inherit the strengths and weaknesses of the underlying distributed shared-memory system. There is no clear consensus regarding the limits of this developing technology. Programming on top of a distributed shared-memory system versus directly programming a distributed-memory system is a trade-off between convenience and control, similar to the trade-offs involved in other aspects of the memory hierarchy. The programmer needs to be aware of the difference in cost between local and remote memory access, and design algorithms with memory access patterns that minimize data transfer between nodes. Explicit copying of data into local memory may still be important in many algorithms.

Chapter 10

Comparison with Related Work

In this chapter, we compare our parallel programming model with related work that is also motivated by the goal of reducing the difficulty of reasoning about explicit parallelism. In particular, we contrast our model with other parallel programming models without multiple threads of control: automatic parallelizing compilers, run-time parallelization systems, parallel declarative programming, and data-parallel programming. We identify the opportunities for integration of constructs and implementation techniques from these models with our parallel programming model. A comparison of our model with other parallel programming models that incorporate similar constructs is given in Chapter 2.

10.1 Overview

The motivation for the design of our parallel programming model is that explicit parallel programs are more difficult to develop than sequential programs due to the complexity of reasoning, testing, and debugging in the context of multiple concurrent threads of control. Our parallel programming model provides a small set of pragmas that indicate how a standard sequential program can be executed as a parallel program. The key to our model is that the pragmas provide direct control of parallelism, yet reasoning, testing, and debugging can be performed in the context of a single sequential thread of control.

We contrast our parallel programming model with three other models that also reduce the difficulty of parallel programming by removing the complexity of multiple concurrent threads of control:

1. Automatic parallelizing compilers.

2. Run-time parallelization systems.

3. Parallel declarative programming.

4. Data-parallel programming.

The major difference between our parallel programming model and these other models is that our model provides explicit control of parallelism and of a sophisticated synchronization mechanism. The intended scope of our parallel programming model is more than traditional high-performance scientific computing. In many aspects, the strengths of these other models are complementary with the strengths of our model and we identify opportunities for integration of constructs and implementation techniques from these models with our parallel programming model.

10.2 Automatic Parallelizing Compilers

A completely automatic parallelizing compiler [12][16][31][32][53][95] transforms a source program written in a sequential language (e.g., Fortran or C) into an equivalent parallel object program for a given target architecture, based on conservative data-dependence and data-flow analysis. The most important transformations are those that allow sequential loops to be converted into parallel loops. The advantages of this approach to parallel programming are obvious:

1. The programmer can write ordinary sequential programs using a familiar notation.

2. Existing sequential programs can be parallelized without additional effort.

The disadvantage is that parallel performance is reliant on the compiler's success at recognizing opportunities for efficient parallel execution in the sequential source program. There are two main reasons why the compiler might be unable to generate an efficient parallel object program:

1. The sequential source program may be efficiently parallelizable using the transformations available to the compiler, but there may be insufficient information in the text of the program for the compiler to recognize the legality of the transformations.

2. The sequential source program may not be efficiently parallelizable using the transformations available to the compiler. An efficient parallel algorithm may require fundamental changes to the data structures and algorithms of the efficient sequential algorithm that are beyond the scope of the compiler.

Most current automatic parallelization techniques detect loop-level parallelism in programs that operate on matrices and produce parallel object code that uses barrier synchronization. A large class of important scientific applications can be efficiently parallelized using this approach. However, the applicability of automatic parallelization has not been convincingly demonstrated for problems that require complicated or dynamically-allocated data structures or that require more complex patterns of parallelism and synchronization.

Some automatic parallelization systems allow the source program to be annotated with extra assertions that can be incorporated in the compiler's data-dependence analysis. For example, annotations might be used to assert that the iterations of a loop do not interfere with each other, that two variables are not aliases of each other, or that a section of code accesses only a certain set of variables. However, as the need for annotations increases, the programmer is required to understand more about the transformations performed by the particular compiler and the advantage of automatic parallelization decreases.

There are two important differences between our pragmas and most compilation systems with annotations: (i) there is an explicit parallel semantics associated with our pragmas, and (ii) the single-assignment pragma provides explicit control over run-time synchronization. Our parallel programming model provides direct control of parallelism and of a sophisticated synchronization mechanism, instead of relying on the transformations available to the compiler. For this reason, our parallel programming model is applicable to a wider range of problems than those for which automatic parallelization is typically successful. A disadvantage of our parallel programming model is that the pragmas can be misused, resulting in erroneous parallel sharing of data, whereas analysis in completely automatic parallelization is conservative and hence safe.

Most of the techniques developed for automatic parallelizing compilers are also applicable to compiling explicitly parallel notations and can be applied to compilation of our parallel programming model. For example, loop restructuring transformations often could be used to reduce the cost of thread creation associated with our parallelizable for-loop statement. A thorough discussion of techniques for generating efficient parallel code from both sequential and parallel notations is given by Wolfe [136]. In addition, our pragmas could be used as program annotations to aid compile-time analysis by an automatic parallelizing compiler.

10.3 Runtime Parallelization Systems

An automatic parallelizing compiler analyzes a sequential source program (possibly with annotations) prior to execution and generates an equivalent parallel object program. An alternative approach is to perform at least part of analysis at run-time when more information is available [103].

Jade [101] is an example of a run-time parallelization system based on program annotations. A Jade program is a standard sequential program with annotations that specify the decomposition of the program into tasks and the data objects accessed by the tasks. At run time, the annotations are evaluated to determine a scheduling of the tasks that is equivalent to sequential execution. The annotations could also be used for compile-time analysis. The validity of the data access annotations can be checked at run time. The basic Jade model consists of identifying independent tasks that can be executed in parallel without synchronization. Jade also includes more sophisticated annotations based on continuations that allow parallel tasks to synchronize their actions.

The motivation behind the Jade system is very similar to that of our parallel programming model. Reasoning, testing, and debugging of a Jade program can be performed in the context of the standard sequential semantics of the underlying programming notation. However, as with our model, it is the responsibility of the programmer to structure and annotate the program to ensure efficient parallel performance.

The major differences between our parallel programming model and Jade are: (i) parallelism is specified directly in our model, whereas data access patterns are specified in Jade and parallelism is extracted by the run time system, and (ii) synchronization between tasks is based on single-assignment variables in our model and on continuations in Jade. Jade has a higher execution overhead than our model because of the cost of evaluating the annotations and determining the scheduling of tasks at run time. However, the cost of checking Jade annotations at run time is less than the cost of checking the restrictions on our pragmas because the Jade annotations directly relate to data accesses.

10.4 Parallel Declarative Programming

A declarative program expresses an algorithm as an expression mapping inputs onto outputs, rather than as a sequence of operations that modify the state of variables. Since there is no concept of program state or flow of control, the subexpressions of a declarative program can be evaluated in any order or in parallel. Declarative languages include functional, logic programming, and dataflow languages. In functional languages [10][35][64][66][91] the mapping between inputs and outputs is expressed using functions. In logic programming languages [30][75] the mapping between inputs and outputs is expressed using relations. Dataflow languages [2][134] are declarative (usually functional) languages designed for execution using the dataflow operational model.

Parallel declarative languages are designed to express declarative programs that are intended to be executed as parallel programs. Examples include the family of parallel functional languages based on Lisp [50][54][68], the family of concurrent logic programming languages based on Prolog [28][110][113], and dataflow languages such as Id [94], Val [88], and Sisal [38]. A review of parallel declarative languages is given by Almasi and Gottlieb [5, Section 5.3] and a collection of papers describing individual parallel functional and dataflow languages was edited by Szymanski [120].

A compiler for a parallel declarative language transforms a declarative source program into an equivalent parallel object program for a given target architecture. The advantages of this approach to parallel programming are:

1. Declarative programming languages express algorithms at a high level of abstraction.

2. The compiler is responsible for recognizing the parallelism that is implicit in the high-level description of the algorithm.

3. The results of parallel functional programs are deterministic, regardless of process mapping and scheduling.

The disadvantages of this approach to parallel programming are:

1. Declarative programming languages have not achieved the popularity of imperative languages in most programming communities.

2. Parallel performance is reliant on the compiler's success at recognizing opportunities for efficient parallel execution.

Proponents of declarative programming [10][67][131] claim that the high level of abstraction decreases program development time and increases reliability and portability. However, declarative languages are not as widely used as traditional imperative languages such as Fortran and C. Whatever the reason for this, the lack of popularity of declarative languages in the sequential domain is a significant hurdle to the acceptance of parallel declarative programming.

At least part of the reason that declarative languages are less popular than imperative languages is the difficulty of compiling declarative programs into efficient code. When compiling to sequential code, sophisticated data-dependence analysis and transformations are required to prevent excessive memory allocation and data copying. Although some aspects of data-dependence analysis are easier because of the high level of algorithm description [17][19][70], automatic parallelization of a declarative program involves many of the same problems as automatic parallelization of a sequential program. Hudak's parafunctional programming approach [63][65] allows a parallel functional program to be annotated to explicitly control granularity and process mapping.

The use of single-assignment variables for synchronization in our parallel programming model is derived from ideas that appear in parallel dataflow and logic programming languages. Many of the programming and compilation techniques developed in the context of parallel declarative languages are applicable to our parallel programming model. The major difference between our model and declarative models is that our parallel programming model provides direct control of which variables are single-assignment and which are mutable and of which statements are executed in parallel and which are executed sequentially. With a declarative language, this is the responsibility of the compiler. In addition, our parallel programming model is integrated with a conventional sequential language, which makes it readily accessible to a larger body of programmers.

10.5 Data-Parallel Programming

A data-parallel program consists of a single thread of control in which each operation can be executed simultaneously on all the elements of a large data set. Data-parallel programming was originally described [58] in the context of programming SIMD (Single Instruction stream, Multiple Data stream) computers such as the Connection Machine [57]. Subsequent work [55][56][100] describes techniques for the implementation of data-parallel programs on MIMD (Multiple Instruction stream, Multiple Data stream) computers. Examples of data-parallel languages include C* [102], Dataparallel C [55], pC++ [80], CM Fortran, Fortran 90 [90], HPF (High Performance Fortran) [40][74], and NESL [14][15].

Parallel operations on the elements of data sets are specified using synchronous parallel loops such as the `FORALL` loop of HPF and other Fortran dialects, and using parallel operators on composite data structures such as the Fortran 90 array intrinsic functions. In some data-parallel languages, e.g., Fortran 90 and HPF, parallel operations are provided only for regular data structures such as dense arrays. In other data-parallel languages, e.g., pC++, parallel operations can also be performed on irregular and user-defined data structures. Some data-parallel languages, notably HPF, provide directives that specify the data distribution and process mapping.

The advantage of data-parallel programming is direct control of parallelism in a simple single-threaded programming model. Since there is a single thread of control, there are no synchronization operations and therefore no possibility of race conditions or deadlock. The major disadvantage of data-parallel programming is the limited range of algorithms that can be expressed. Although, efficient data-parallel algorithms have been demonstrated for wide range of applications [69], many efficient parallel algorithms involve multiple threads of control executing different instruction streams. Some multithreaded algorithms can be expressed by extending the data-parallel programming model to allow nested data-parallel constructs, as in NESL. A more general approach is to integrate data-parallel programming with task-parallel programming, as in several proposed extensions to HPF [25][41][52].

When used without single-assignment types, our parallel programming model is very similar to the nested data-parallel model. The difference is that our model allows synchronization between multiple threads of control and can therefore express algorithms that cannot be expressed using the nested data-parallel model. Many features of data-parallel languages, such as annotations for data distribution and parallel operations on composite data structures could be added to our parallel programming model. Techniques that have been developed for compiling data-parallel loops could be applied to some instances of our parallelizable for-loop statement.

Chapter 11

Conclusions

11.1 Summary

The problem that we have addressed in this thesis is the difficulty of developing parallel programs as compared to equivalent sequential programs due to the complexity of reasoning, testing, and debugging in the context of multiple concurrent threads of control. We have presented and investigated a parallel programming model that consists of a standard sequential notation extended with three pragmas:

1. The parallelizable sequence of statements pragma is used to indicate that a sequence of statements can be executed in parallel.

2. The parallelizable for-loop statement pragma is used to indicate that the iterations of a for-loop statement can be executed in parallel.

3. The single-assignment type pragma is used to indicate that the variables of a given type are assigned at most once and that assignment and evaluation operations on those variables can be used as synchronization operations between parallel threads.

We have proved that if the placement of the pragmas satisfies a small set of restrictions, execution of a program according to the parallel semantics is equivalent to execution of the program according to the standard sequential semantics. Our model allows parallel programs to be developed using reasoning, testing, and debugging in the context of a single thread of control, yet it provides direct specification of parallel execution and synchronization.

We have performed a series of programming experiments designed to investigate the expressiveness and efficiency of our parallel programming model and to assess the merits of our parallel program development methodology. These experiments involved combinatorial and numerical problems, and used arrays and dynamically-allocated data structures. We have presented performance measurements for the execution of these experimental parallel programs on up to 32 processors of a shared-memory symmetric multiprocessor system. Parallel speedups were presented relative to the execution of efficient sequential programs on a single processor.

We have identified limitations on the expressiveness of our parallel programming model. In particular, our model cannot express parallel algorithms in which the actions of the computation are nondeterministically affected by the timing of the execution of the parallel threads. We have discussed issues involved in the implementation of our parallel programming model on distributed memory computer systems and we have identified opportunities for the integration of our model with related work that provides complementary strengths and additional functionality.

11.2 Findings

11.2.1 Expressiveness

In each of our programming experiments, we found that the most efficient parallel algorithm we know could be elegantly expressed using our programming model. Although the parallel programs involve sophisticated control and synchronization patterns, they are not much more complicated than efficient sequential programs, and are less complicated than efficient parallel programs expressed using less-structured parallel programming models, e.g., thread libraries providing barrier and lock synchronization.

We found that the forms of the parallelizable for-loop statement with pattern arguments provide a concise means of expressing control of the level of granularity of a program, without adding extra complexity to the algorithm. In our programming experiments, almost every parallelizable for-loop statement required arguments for efficiency. This is because the number of iterations of a for-loop statement usually does not represent the most efficient level of granularity for parallelism in a program. All three patterns for assigning iterations to parallel threads were found to be useful in different situations. In a model without equivalent parallel and sequential semantics, a parallel for-loop statement could not be parameterized in this manner without risk of deadlock.

We found that single-assignment types provide a high-level approach to expressing sophisticated synchronization patterns based on data-flow, with very little notational overhead compared to equivalent sequential algorithms. Single-assignment types implicitly combine mutual exclusion and broadcast synchronization through ordinary assignment and evaluation operations. Our programming experiments demonstrate that integrating single-assignment types with the type system of the underlying language allows single-assignment variables to be embedded in ordinary data structures to provide synchronization at an efficient level of granularity.

We observe that the full generality of single-assignment types is not necessary except for very fine-grained parallel programs. To control the level of granularity, single-assignment variables usually need to be distinct components the data structures on which they synchronize operations. This is because the number of components of a data-structure usually does not represent the most efficient level of granularity for synchronization in a program. In most cases, the same functionality can be obtained using single-assignment unary types with set and check operations. We refer to this synchronization type as a "broadcast flag".

Adding broadcast flags to a sequential notation as a predefined type is considerably more simple than integrating single-assignment types with the entire type system. In addition, algorithms described in terms of broadcast flags are more easily portable between different notations than algorithms described in terms of single-assignment types, because the semantics of single-assignment types may vary depending on the details of the underlying type system.

The most significant limitation that we found on the expressiveness of our parallel programming model is that our model cannot express algorithms in which the actions of the computation are nondeterministically affected by the timing of the execution of the parallel threads. Many efficient parallel search algorithms require this kind of timing-dependent nondeterminacy and therefore cannot be expressed using our parallel programming model.

11.2.2 Efficiency

In each of our performance experiments, we found that our parallel programs executed as fast as efficient sequential programs on a single processor and delivered good speedups on up to 32 processors of a shared-memory symmetric multiprocessor system. These performance measurements demonstrate that single-assignment types (or broadcast flags) can be an efficient synchronization mechanism for coarse-grained to medium-grained parallel programs on small-scale to moderate-scale multiprocessor systems.

We have not yet investigated the question of how efficiently single-assignment types (or broadcast flags) can be implemented as low-level library or hardware primitives to support very fine-grained parallel programming. It would be useful to determine whether single-assignment types have any significant advantages or disadvantages in terms of efficiency compared to other synchronization primitives, e.g., locks, when supported at this level.

We have not yet investigated the question of how efficiently access to remote single-assignment variables (or broadcast flags) can be implemented on a distributed-memory computer system. The straightforward approach is to implement every access to a remote single-assignment variable as a series of communication operations. It would be useful to design efficient algorithms for automatic local caching of remote single-assignment variables that take advantage of the property that a cached and assigned single-assignment variable will never be invalidated.

11.2.3 Development Methodology

From our programming experiments, we are convinced of the merits of our program development methodology based on the equivalence of the parallel and sequential semantics. All of our experimental programs were developed, tested, and debugged as sequential programs, then later translated into parallel programs as directed by the pragmas, for parallel testing and performance measurement. Minor errors were discovered and corrected during the sequential phase of development, but no additional errors were discovered during parallel testing and performance measurement. Because of the sophistication of the parallel control and synchronization patterns, we believe that it would have been significantly more difficult to develop these programs using a less-structured parallel programming model, e.g., a thread library providing barrier and lock synchronization.

The greatest potential for subtle errors in our parallel programming model results from the restrictions on: (i) parallel access to shared variables, and (ii) access to variables following exceptional termination of a parallelizable construct. In general, these errors cannot be caught either at compile time or at run time with reasonable cost. However, in our programming experiments, we found that these errors were relatively easy to avoid. In practice, subtle synchronization of operations on shared variables is usually localized. In practice, exceptions are usually either handled in the block in which they are raised or else they are left to propagate to the program level as run-time errors.

To help with the avoidance of errors, an area for further investigation is the extension of assertional specification and reasoning systems to support our parallel programming model. The public specification of a subprogram must somehow represent the operations on shared private variables that occur in its implementation so that subprograms can be composed on the basis of their specifications without violating the restrictions on our model. To help with the detection of errors, an area for further investigation is the development of efficient algorithms for run-time detection of erroneous execution situations during either sequential or parallel execution.

11.2.4 Integration with Other Models

Although we are enthusiastic regarding its strengths, we do not believe that our model provides the best solution to all parallel programming problems. Our model cannot express parallel algorithms in which the actions of the computation are nondeterministically affected by the timing of the execution of the parallel threads. For this reason, we recommend that any practical implementation of our parallel programming model should be integrated with a framework that also provides less-structured thread creation and synchronization constructs. We have outlined one approach to the integration of our model with less-structured models.

In addition, there are other important aspects of a complete parallel programming system to which we have not contributed in this research, where our model could be integrated with methods and constructs developed in other contexts. Data dependence analysis and code restructuring techniques developed in the context of parallel declarative languages and automatic parallelizing compilers could be applied to the efficient compilation of our model. Methods for implementing shared-memory models on top of distributed-memory computer systems could be used to provide distributed-memory implementations of our model. Data distribution annotations, automatic process mapping techniques, and language constructs from data-parallel programming could be added to our model.

Appendix A

Transformation of Pragmas to Ada 95 Tasking Constructs

In this appendix, we present the transformations from our pragmas to standard Ada 95 tasking constructs that we use to implement the parallel semantics of our programming model for our parallel programming performance experiments. For each of the pragmas, we give the transformations both with parallel runtime checks and with parallel runtime checks suppressed.

A.1 Parallelizable Sequence of Statements

A parallelizable sequence of statements:

pragma Parallelizable_Sequence;
$\mathsf{statement}_1$
...
$\mathsf{statement}_n$

is transformed into a block statement containing a sequence of task declarations, with one task declaration for each statement.

A.1.1 Transformation with Parallel Exception Checking Suppressed

A parallelizable sequence of statements, with parallel exception checking suppressed, is transformed into the following:

```
declare
    task T_1; task body T_1 is
    begin
        statement_1
    end T_1;
    ...
    task T_N; task body T_N is
    begin
        statement_n
    end T_N;
begin
    null;
end;
```

A.1.2 Transformation without Parallel Exception Checking Suppressed

A parallelizable sequence of statements, without parallel exception checking suppressed, is transformed into the following:

```
use Ada.Exceptions;

declare
    Join : Thread_Join (n);

    task T_1; task body T_1 is
    begin
        statement_1
        Join.Thread_Complete(Error => Null_Id);
    exception
        when E : others =>
            Join.Thread_Complete(Error => Exception_Identity(E));
    end T_1;
    ...
    task T_N; task body T_N is
    begin
        statement_n
        Join.Thread_Complete(Error => Null_Id);
    exception
        when E : others =>
            Join.Thread_Complete(Error => Exception_Identity(E));
    end T_N;

    Error : Exception_Id;
begin
    Join.All_Complete(Error);
    if Error /= Null_Id then
        abort T_1;
        ...
        abort T_N;
        Raise_Exception(Error);
    end if;
end;
```

A.2 Parallelizable For-Loop Statement

A parallelizable for-loop statement without arguments:

```
pragma Parallelizable_Loop;
for I in [reverse] loop_range loop
    loop_body
end loop;
```

is transformed into a block statement containing an array of tasks, with one task for each iteration.

A.2.1 Transformation with Parallel Exception Checking Suppressed

A parallelizable sequence of statements, with parallel exception checking suppressed, is transformed into the following:

```
declare
    task type Iteration (I : Loop_Range);
    task body Iteration is
    begin
        loop_body
    end Iteration;

    type Iteration_Access is access Iteration;

    Iterations : array (Loop_Range) of Iteration_Access;
begin
    for I in loop_range loop
        Iterations(I) := new Iteration (I);
    end loop;
end;
```

A.2.2 Transformation without Parallel Exception Checking Suppressed

A parallelizable sequence of statements, with parallel exception checking suppressed, is transformed into the following:

```
declare
    Num_Iterations : constant Integer :=
        Integer'Max(0, Loop_Range'Pos(Loop_Range'Last) −
                       Loop_Range'Pos(Loop_Range'First) + 1);

    Join : Thread_Join (Num_Iterations);

    task type Iteration (I : Loop_Range);
    task body Iteration is
    begin
        loop_body
        Join.Thread_Complete(Error => Null_Id);
    exception
        when E : others =>
            Join.Thread_Complete(Error => Exception_Identity(E));
    end Iteration;

    type Iteration_Access is access Iteration;

    Iterations : array (Loop_Range) of Iteration_Access;
    Error      : Exception_Id;
begin
    for I in loop_range loop
        Iterations(I) := new Iteration (I);
    end loop;
    Join.All_Complete(Error);
    if Error /= Null_Id then
        for I in Loop_Range loop
            abort Iterations(I).all;
        end loop;
        Raise_Exception(Error);
    end if;
end;
```

A.3 A Protected Type for Joining Parallel Threads

```
use Ada.Exceptions;

protected Thread_Join (Num_Threads : Integer) is
   procedure Thread_Complete (Error : in Exception_Id);
   entry All_Complete (Error : out Exception_Id);
private
   Completed       : Boolean := Num_Threads <= 0;
   Num_Complete    : Integer := 0;
   Propagated_Error : Exception_Id := Null_Id;
end Thread_Join;

protected body Thread_Join is
   procedure Thread_Complete (Error : in Exception_Id) is
   begin
      if Error /= Null_Id then
         Completed := True;
         Propagated_Error := Error;
      end if;
      Num_Complete := Num_Complete + 1;
      if Num_Complete = N then
         Completed := True;
      end if;
   end Thread_Complete;
   entry All_Complete (Error : out Exception_Id) when Completed is
   begin
      Error := Propagated_Error;
   end All_Complete;
end Thread_Join;
```

A.4 Single-Assignment Types

A.4.1 Transformation of a Type Declaration

A single-assignment type declaration:

```
type S is type_definition;
pragma Single_Assignment(S);
```

is transformed into a record type consisting of the mutable type and an associated protected type that synchronizes assignment and evaluation operations.

```
package S is
    type Value is type_definition;
    type Variable is limited private;
    procedure Assign (To : in out Variable; From : in Value);
    procedure Unchecked_Assign (To : in out Variable; From : in Value);
    function Read (From : Variable) return Value;
private
    type Variable is
        record
            Contents : Value;
            Guard    : Single_Assignment_Guard;
        end record;
end S;

package body S is
    procedure Assign (To : in out Variable; From : in Value) is
        Previously_Set : Boolean;
    begin
        To.Guard.Test_And_Lock(Previously_Set);
        if Previously_Set then
            raise Constraint_Error;
        else
            To.Contents := From;
            To.Guard.Set_And_Unlock;
        end if;
    end Assign;

    procedure Unchecked_Assign (To : in out Variable; From : in Value) is
    begin
        To.Contents := From;
        To.Guard.Set;
    end Unchecked_Assign;

    function Read (From : Variable) return Value is
    begin
        From.Guard.Wait_Until_Set;
        return From.Contents;
    end Read;
end S;
```

A.4.2 Transformation of Operations on Variables

Assignment and evaluation operations on single-assignment variables:

```
X := Y;
```

(where X and Y are variables of a single-assignment type S) are transformed to calls to the Assign and Read subprograms of the transformed single-assignment type declaration.

If single-assignment checks are suppressed, the transformation is into the following:

```
S.Unchecked_Assign(To => X, From => S.Read(Y));
```

If single-assignment checks are suppressed, the transformation is into the following:

```
S.Assign(To => X, From => S.Read(Y));
```

A.4.3 A Protected Type for Synchronizing Operations

```
protected type Single_Assignment_Guard is
    entry Test_And_Lock (Set : out Boolean);
    procedure Set;
    procedure Set_And_Unlock;
    entry Wait_Until_Set;
private
    Is_Locked : Boolean := False;
    Is_Set    : Boolean := False;
end Single_Assignment_Guard;

protected body Single_Assignment_Guard is
    entry Test_And_Lock (Set : out Boolean) when not Is_Locked is
    begin
        Set := Is_Set;
        Is_Locked := not Is_Set;
    end Test_And_Lock;
    procedure Set is
    begin
        Is_Set := True;
    end Set;
    procedure Set_And_Unlock is
    begin
        Is_Set := True;
        Is_Locked := False;
    end Set_And_Unlock;
    entry Wait_Until_Set when Is_Set is
    begin
        null;
    end Wait_Until_Set;
end Single_Assignment_Guard;
```

Appendix B

Experimental Programs

In this appendix, we present the complete text of the experimental programs that we describe in chapters 6 and 7. We give both the parallel programs and the sequential programs that are used for speedup comparisons.

B.1 One-Deep Parallel Mergesort

Declaration of Sequential Quicksort

```
------------------------------------------------------------------------
-- Generic Standard Quicksort Package Declaration
------------------------------------------------------------------------

generic
   type Element is private;
   type Elements is array (Integer range <>) of Element;
   with function "<" (Left, Right : Element) return Boolean is <>;
   with function ">" (Left, Right : Element) return Boolean is <>;
   with function "=" (Left, Right : Element) return Boolean is <>;
   with function "<=" (Left, Right : Element) return Boolean is <>;
   with function ">=" (Left, Right : Element) return Boolean is <>;
package Standard_Quicksort is

   procedure Sequential_Quicksort (
         Data        : in out Elements;
         Base_Length : in     Positive );
   --| Requires:
   --|    2 <= Base_Length.
   --| Ensures:
   --|    Ascending(Data) and Permutation(Data, in Data).

end Standard_Quicksort;

------------------------------------------------------------------------
```

Definition of Sequential Quicksort

```
----------------------------------------------------------------------
-- Generic Standard Quicksort Package Body
--
-- Algorithm details:
--
-- * Recursive sequential quicksort algorithm.
-- * Base-case data arrays sorted using insertion sort.
-- * Median of first, middle, and last elements used as
--   pivot in partitioning.
-- * Sentinels used to reduce number of comparisons the partitioning.
-- * Source: "Numerical Recipes in C, Second Edition".
----------------------------------------------------------------------

with Assertions; use Assertions;

package body Standard_Quicksort is

   -------------------------------------------------------------------

   procedure Insertion_Sort (Data : in out Elements) is
   --| Ensures:
   --|    Ascending(Data) and Permutation(Data, in Data).
   begin
      for I in Data'First .. Data'Last - 1 loop
         declare
            Temp : Element;
            Pos  : Integer range Data'Range;
         begin
            Temp := Data(I + 1);
            Pos  := Data'First;
            for J in reverse Data'First .. I loop
               if Temp < Data(J) then
                  Data(J + 1) := Data(J);
               else
                  Pos := J + 1;
                  exit;
               end if;
            end loop;
            Data(Pos) := Temp;
         end;
      end loop;
   end Insertion_Sort;

   -------------------------------------------------------------------

   procedure Swap (X, Y : in out Element) is
   --| Ensures:
   --|    X = in Y and Y = in X.

      Temp : Element;

   begin
      Temp := X;
      X := Y;
```

```
   Y := Temp;
end Swap;

----------------------------------------------------------------------

procedure Partition (
      Data        : in out Elements;
      Pivot_Index :    out Integer  ) is
--| Requires:
--|    Data'Length >= 3.
--| Ensures:
--|    Data'First <= Pivot_Index and Pivot_Index <= Data'Last and
--|    for all I in Data'First .. Pivot_Index - 1 :
--|       Data(I) <= Data(Pivot_Index) and
--|    for all I in Pivot_Index + 1 .. Data'Last :
--|       Pivot_Index <= Data(I).

   First  : constant Integer := Data'First;
   Last   : constant Integer := Data'Last;
   Length : constant Integer := Last - First + 1;
   Pivot_Value : Element;
   Left, Right : Integer range Data'Range;

begin
   Assert(Data'Length >= 3);
   Swap(Data(First + 1), Data((First + Last)/2));
   if Data(First + 1) > Data(Last) then
      Swap(Data(First + 1), Data(Last));
   end if;
   if Data(First) > Data(Last) then
      Swap(Data(First), Data(Last));
   end if;
   if Data(First + 1) > Data(First) then
      Swap(Data(First + 1), Data(First));
   end if;
   Pivot_Value := Data(First);
   Left := First + 1; Right := Last;
   loop
      Left := Left + 1;
      while Data(Left) < Pivot_Value loop
         Left := Left + 1;
      end loop;
      Right := Right - 1;
      while Data(Right) > Pivot_Value loop
         Right := Right - 1;
      end loop;
      exit when Right < Left;
      Swap(Data(Left), Data(Right));
   end loop;
   Data(First) := Data(Right);
   Data(Right) := Pivot_Value;
   Pivot_Index := Right;
   Assert(Data'First <= Pivot_Index and Pivot_Index <= Data'Last);
end Partition;
```

```
   ----------------------------------------------------------------------

   procedure Sequential_Quicksort (
         Data        : in out Elements;
         Base_Length : in     Positive ) is
   begin
      Assert(2 <= Base_Length);
      if Data'Length <= Base_Length then
         Insertion_Sort(Data);
      else
         declare
            Pivot_Index : Integer range Data'Range;
         begin
            Partition(Data, Pivot_Index);
            Sequential_Quicksort(
               Data(Data'First .. Pivot_Index - 1), Base_Length);
            Sequential_Quicksort(
               Data(Pivot_Index + 1 .. Data'Last), Base_Length);
         end;
      end if;
   end Sequential_Quicksort;

   ----------------------------------------------------------------------

end Standard_Quicksort;

-------------------------------------------------------------------------
```

Declaration of One-Deep Parallel Mergesort

```
------------------------------------------------------------------------
-- Generic One-Deep Parallel Mergesort Package Declaration
------------------------------------------------------------------------

generic
   type Element is private;
   type Elements is array (Integer range <>) of Element;
   with function "<" (Left, Right : Element) return Boolean is <>;
   with function ">" (Left, Right : Element) return Boolean is <>;
   with function "=" (Left, Right : Element) return Boolean is <>;
   with function "<=" (Left, Right : Element) return Boolean is <>;
   with function ">=" (Left, Right : Element) return Boolean is <>;
package One_Deep_Mergesort is

   procedure Parallel_Mergesort (
         Data        : in out Elements;
         Result      :    out Elements;
         Num_Threads : in     Positive );
   --| Requires:
   --|    Data'Range = Result'Range and
   --|    2*Num_Threads*Num_Threads <= Data'Length.
   --| Ensures:
   --|    Ascending(Result) and Permutation(result, in Data).

end One_Deep_Mergesort;

------------------------------------------------------------------------
```

Definition of One-Deep Parallel Mergesort

```
----------------------------------------------------------------------
-- Generic One-Deep Parallel Mergesort Package Body
----------------------------------------------------------------------

with Assertions; use Assertions;
with Standard_Quicksort;

package body One_Deep_Mergesort is

   -------------------------------------------------------------------

   Base_Length : constant Positive := 16;

   -------------------------------------------------------------------

   type Indices is array (Integer range <>) of Integer;

   type Segment is
      record
         First, Last : Integer;
      end record;
   type Segments is array (Integer range <>) of Segment;

   -------------------------------------------------------------------

   package Element_Standard_Quicksort is
      new Standard_Quicksort (Element, Elements);
   use Element_Standard_Quicksort;

   -------------------------------------------------------------------

   package Index_Standard_Quicksort is
      new Standard_Quicksort (Integer, Indices);
   use Index_Standard_Quicksort;

   -------------------------------------------------------------------

   function Split (
         First, Last, I : Integer; N : Positive) return Integer is
   begin
      return First +
         Integer(Float(Last - First + 1)*(Float(I)/Float(N)));
   end Split;

   -------------------------------------------------------------------

   function Search (Data : Elements; Target : Element) return Integer is
   --| Requires:
   --|    Ascending(Data).
   --| Ensures:
   --|    (Search = Data'First or else Data(Search - 1) < Target) and
   --|    (Search = Data'Last + 1 or else Target <= Data(Search)).

      F, L, Mid : Integer range Data'First - 1 .. Data'Last + 1;
```

```
begin
   F := Data'First; L := Data'Last;
   while F <= L loop
      Mid := (L - F)/2 + F;
      if Data(Mid) < Target then
         F := Mid + 1;
      else
         L := Mid - 1;
      end if;
   end loop;
   Assert(F = L + 1);
   Assert(F = Data'First or else Data(F - 1) < Target);
   Assert(F = Data'Last + 1 or else Target <= Data(F));
   return F;
end Search;

----------------------------------------------------------------------

procedure Sequential_Multiway_Merge (
      Data   : in     Elements;
      P      : in     Positive;
      S      : in     Segments;
      Merged :    out Elements ) is
--| Requires:
--|    S'First = 0 and S'Last = P - 1 and
--|    for all I in 0 .. P - 1 :
--|      (Data'First <= S(I).First and S(I).Last <= Data'Last and
--|       Ascending(Data(S(I).First .. S(I).Last))) and
--|    Merged'Length =
--|       Sum(I in 0 .. P - 1 : S(I).Last - S(I).First + 1).
--| Ensures:
--|    Ascending(Merged) and
--|    Permutation(Merged,
--|       Join(I in 0 .. P - 1 : Data(S(I).First .. S(I).Last))).

   Max_Index : Integer range Data'Range;

   Next : array (Integer range 0 .. P - 1) of
             Integer range Data'First .. Data'Last;
   Heap : array (Integer range 0 .. P - 1) of
             Integer range 0 .. P - 1;

begin
   Assert(S'First = 0 and S'Last = P - 1);
   if Data'Length > 0 then
      declare
         Total       : Natural;
         Max_Element : Element;
         Found       : Boolean;
      begin
         Total := 0;
         Found := False;
         for I in 0 .. P - 1 loop
            Assert(Data'First <= S(I).First);
```

```
      Assert(S(I).Last <= Data'Last);
      Total := Total + S(I).Last - S(I).First + 1;
      if S(I).Last >= S(I).First and then (not Found
            or else Data(S(I).Last) > Max_Element) then
         Max_Index := S(I).Last;
         Max_Element := Data(Max_Index);
         Found := True;
      end if;
   end loop;
   Assert(Merged'Length = Total);
end;
for I in 0 .. P - 1 loop
   declare
      Pos : Integer range 0 .. P - 1;
   begin
      if S(I).First <= S(I).Last then
         Next(I) := S(I).First;
      else
         Next(I) := Max_Index;
      end if;
      Pos := 0;
      for J in reverse 0 .. I - 1 loop
         if Data(Next(Heap(J))) > Data(Next(I)) then
            Heap(J + 1) := Heap(J);
         else
            Pos := J + 1;
            exit;
         end if;
      end loop;
      Heap(Pos) := I;
   end;
end loop;
for I in Merged'Range loop
   declare
      Top   : Integer range 0 .. P - 1;
      Child : Integer range 0 .. 2*P;
   begin
      Top := Heap(0);
      Merged(I) := Data(Next(Top));
      if Next(Top) = S(Top).Last then
         Next(Top) := Max_Index;
      elsif Next(Top) /= Max_Index then
         Next(Top) := Next(Top) + 1;
      end if;
      Child := 1;
      while Child < P loop
         if Child < P - 1 and then
               Data(Next(Heap(Child + 1)))
               < Data(Next(Heap(Child))) then
            Child := Child + 1;
         end if;
         exit when Data(Next(Top)) <= Data(Next(Heap(Child)));
         Heap((Child - 1)/2) := Heap(Child);
         Child := 2*Child + 1;
      end loop;
```

```
            Heap((Child - 1)/2) := Top;
         end;
      end loop;
   end if;
end Sequential_Multiway_Merge;

---------------------------------------------------------------------

procedure Parallel_Multiway_Merge (
      Data   : in     Elements;
      P      : in     Positive;
      D      : in     Indices;
      Result :    out Elements ) is
--| Requires:
--|    Data'Range = Result'Range and D'Range = 0 .. P and
--|    2*P*P <= Data'Length and Partition(D, Data'Range) and
--|    for all I in 0 .. P : Ascending(Data(D(I) .. D(I + 1) - 1)).
--| Ensures:
--|    Ascending(Result) and Permutation(Result, Data).

   Pivot   : Elements (0 .. 2*P*P - 1);
   Index   : Integer range Data'First - 1 .. Data'Last + 1;
   Segment : array (0 .. P - 1) of Segments (0 .. P - 1);
   R       : Indices (0 .. P);

begin
   Assert(Data'First = Result'First and Data'Last = Result'Last);
   Assert(D'First = 0 and D'Last = P);
   Assert(2*P*P <= Data'Length);

   for I in 0 .. P - 1 loop
      Pivot(2*P*I) := Data(D(I));
      for J in 1 .. P - 1 loop
         Index := Split(D(I), D(I + 1) - 1, J, P);
         Pivot(2*P*I + 2*J) := Data(Index);
         Pivot(2*P*I + 2*J - 1) := Data(Index - 1);
      end loop;
      Pivot(2*P*(I + 1) - 1) := Data(D(I + 1) - 1);
   end loop;

   Sequential_Quicksort(Pivot, Base_Length);

   for I in 0 .. P - 1 loop
      Pivot(I) := Pivot(Split(Pivot'First, Pivot'Last, I, P));
   end loop;

   for I in 0 .. P - 1 loop
      Segment(0)(I).First := D(I);
      for J in 1 .. P - 1 loop
         Segment(J)(I).First :=
            Search(Data(D(I) .. D(I + 1) - 1), Pivot(J));
         Segment(J - 1)(I).Last := Segment(J)(I).First - 1;
      end loop;
      Segment(P - 1)(I).Last := D(I + 1) - 1;
   end loop;
```

```
      R(0) := Result'First;
      for I in 1 .. P - 1 loop
         R(I) := R(I - 1);
         for J in 0 .. P - 1 loop
            R(I) := R(I) +
               Segment(I - 1)(J).Last - Segment(I - 1)(J).First + 1;
         end loop;
      end loop;
      R(P) := Result'Last + 1;

      pragma Parallelizable_Loop;
      for I in 0 .. P - 1 loop
         Sequential_Multiway_Merge(
            Data, P, Segment(I), Result(R(I) .. R(I + 1) - 1));
      end loop;
   end Parallel_Multiway_Merge;

   ---------------------------------------------------------------------

   procedure Parallel_Mergesort (
         Data        : in out Elements;
         Result      :    out Elements;
         Num_Threads : in     Positive ) is

      D : Indices (0 .. Num_Threads);

   begin
      for I in 0 .. Num_Threads loop
         D(I) := Split(Data'First, Data'Last, I, Num_Threads);
      end loop;
      pragma Parallelizable_Loop;
      for I in 0 .. Num_Threads - 1 loop
         Sequential_Quicksort(
            Data(D(I) .. D(I + 1) - 1), Base_Length);
      end loop;
      Parallel_Multiway_Merge(Data, Num_Threads, D, Result);
   end Parallel_Mergesort;

   ---------------------------------------------------------------------

end One_Deep_Mergesort;

------------------------------------------------------------------------
```

B.2 The Paraffins Problem

B.2.1 Sequential Solution to the Paraffins Problem

Declaration of Sequential Solution to the Paraffins Problem

```
------------------------------------------------------------------------
-- Paraffins_Problem Package Declaration (Sequential)
------------------------------------------------------------------------

package Paraffins_Problem is

   type Radical_Kind is (Hydrogen, Carboniferous);
   type Radical;
   type Radical_Pointer is access all Radical;
   type Radical_Pointers is array (Integer range <>) of Radical_Pointer;
   type Radical is
      record
         Kind  : Radical_Kind;
         Bonds : Radical_Pointers (1 .. 3);
      end record;
   type Radical_Array is array (Integer range <>) of aliased Radical;
   type Radical_Array_Pointer is access Radical_Array;
   type Radical_Array_Pointers is
      array (Natural range <>) of Radical_Array_Pointer;

   type Paraffin_Kind is (Bond_Centered, Carbon_Centered);
   type Paraffin is
      record
         Kind  : Paraffin_Kind;
         Bonds : Radical_Pointers (1 .. 4);
      end record;
   type Paraffin_Array is array (Integer range <>) of Paraffin;
   type Paraffin_Array_Pointer is access Paraffin_Array;
   type Paraffin_Array_Pointers is
      array (Positive range <>) of Paraffin_Array_Pointer;

   procedure Deallocate (Radicals : in out Radical_Array_Pointers);

   procedure Deallocate (Paraffins : in out Paraffin_Array_Pointers);

   procedure Generate_Paraffins (
         Radicals  :    out Radical_Array_Pointers;
         Paraffins :    out Paraffin_Array_Pointers);
   --| Requires:
   --|    Radicals'First = 0 and Radicals'Last = Paraffins'Last/2.
   --| Ensures:
   --|    for all R in Radicals'Range :
   --|       All_Radicals_Of_Size(R, Radicals) and
   --|    for all P in Paraffins'Range :
   --|       All_Paraffins_Of_Size(P, Paraffins).

end Paraffins_Problem;

------------------------------------------------------------------------
```

Definition of Sequential Solution to the Paraffins Problem

```
----------------------------------------------------------------------
-- Paraffins_Problem Package Body (Sequential)
----------------------------------------------------------------------

with Assertions; use Assertions;
with Ada.Unchecked_Deallocation;

package body Paraffins_Problem is

   -------------------------------------------------------------------

   procedure Deallocate is
      new Ada.Unchecked_Deallocation (
         Object => Radical_Array, Name => Radical_Array_Pointer);

   -------------------------------------------------------------------

   procedure Deallocate is
      new Ada.Unchecked_Deallocation (
         Object => Paraffin_Array, Name => Paraffin_Array_Pointer);

   -------------------------------------------------------------------

   procedure Deallocate (Radicals : in out Radical_Array_Pointers) is
   begin
      for R in Radicals'Range loop Deallocate(Radicals(R)); end loop;
   end Deallocate;

   -------------------------------------------------------------------

   procedure Deallocate (Paraffins : in out Paraffin_Array_Pointers) is
   begin
      for P in Paraffins'Range loop Deallocate(Paraffins(P)); end loop;
   end Deallocate;

   -------------------------------------------------------------------

   function Min (X, Y : Integer) return Integer renames Integer'Min;

   -------------------------------------------------------------------

   type Bonded_Radical_Sizes is array (Positive range <>) of Natural;

   function Num_Arrangements (
         Radicals : Radical_Array_Pointers;
         Sizes    : Bonded_Radical_Sizes   ) return Positive is

      Length      : Positive;
      Numerator   : Positive;
      Denominator : Positive;
      Count       : Natural;

   begin
      Numerator := Radicals(Sizes(Sizes'First))'Length;
```

```
   Denominator := 1;
   Count := 0;
   for R in Sizes'First + 1 .. Sizes'Last loop
      Assert(Sizes(R - 1) <= Sizes(R));
      if Sizes(R) = Sizes(R - 1) then
         Count := Count + 1;
      else
         Count := 0;
      end if;
      Length := Radicals(Sizes(R))'Length;
      Numerator := Numerator*(Length + Count);
      Denominator := Denominator*(Count + 1);
   end loop;
   return Numerator/Denominator;
end Num_Arrangements;

---------------------------------------------------------------------

function Num_Radicals (
      Size     : Natural;
      Radicals : Radical_Array_Pointers) return Positive is

   Result : Natural;

begin
   Assert(Radicals'First = 0 and Size - 1 <= Radicals'Last);
   if Size = 0 then
      Result := 1;
   else
      Result := 0;
      for I in (Size + 1)/3 .. Size - 1 loop
         for J in (Size - I)/2 .. Min(I, Size - 1 - I) loop
            Result := Result + Num_Arrangements(
               Radicals, (1 => Size - 1 - I - J, 2 => J, 3 => I));
         end loop;
      end loop;
   end if;
   return Result;
end Num_Radicals;

---------------------------------------------------------------------

function Num_Paraffins (
      Size     : Positive;
      Radicals : Radical_Array_Pointers) return Positive is

   Result : Natural;

begin
   Assert(Radicals'First = 0 and Size/2 <= Radicals'Last);
   Result := 0;
   if Size mod 2 = 0 then
      Result := Result +
         Radicals(Size/2)'Length*(Radicals(Size/2)'Length + 1)/2;
   end if;
```

```
   for I in (Size + 2)/4 .. (Size - 1)/2 loop
      for J in (Size + 1 - I)/3 .. Min(I, Size - 1 - I) loop
         for K in (Size - I - J)/2 .. Min(J, Size - 1 - I - J) loop
            Result := Result + Num_Arrangements(Radicals,
               (1 => Size - 1 - I - J - K, 2 => K, 3 => J, 4 => I));
         end loop;
      end loop;
   end loop;
   return Result;
end Num_Paraffins;

---------------------------------------------------------------------

function Last_Index (
      Outer_Size, This_Size  : Natural;
      Outer_Index, This_Last : Integer ) return Integer is
begin
   if Outer_Size = This_Size then
      return Outer_Index;
   else
      return This_Last;
   end if;
end Last_Index;

---------------------------------------------------------------------

procedure Generate_Radicals_Of_Shape (
      Result                 : in     Radical_Array_Pointer;
      Last                   : in out Natural;
      Size_1, Size_2, Size_3 : in     Natural;
      Radicals               : in     Radical_Array_Pointers) is
begin
   Assert(Result /= null);
   Assert(Result'First - 1 <= Last and Last <= Result'Last - 1);
   Assert(Size_1 <= Size_2 and Size_2 <= Size_3);
   Assert(Radicals'First <= Size_3 and Size_1 <= Radicals'Last);
   for I in 1 .. Radicals(Size_1)'Last loop
      for J in 1 .. Last_Index(
            Size_1, Size_2, I, Radicals(Size_2)'Last) loop
         for K in 1 .. Last_Index(
               Size_2, Size_3, J, Radicals(Size_3)'Last) loop
            Last := Last + 1;
            Result(Last) := (Carboniferous,
               (Radicals(Size_1)(I)'Access,
                Radicals(Size_2)(J)'Access,
                Radicals(Size_3)(K)'Access ));
         end loop;
      end loop;
   end loop;
end Generate_Radicals_Of_Shape;

---------------------------------------------------------------------

procedure Generate_Radicals_Of_Size (
      Result   :    out Radical_Array_Pointer;
```

```
      Size     : in     Natural;
      Radicals : in     Radical_Array_Pointers) is

   Length : Positive;
   Last   : Natural;

begin
   Assert(Radicals'First = 0 and Size - 1 <= Radicals'Last);
   Length := Num_Radicals(Size, Radicals);
   Result := new Radical_Array (1 .. Length);
   Last := 0;
   if Size = 0 then
      Last := Last + 1;
      Result(Last) := (Hydrogen, (null, null, null));
   else
      for I in (Size + 1)/3 .. Size - 1 loop
         for J in (Size - I)/2 .. Min(I, Size - 1 - I) loop
            Generate_Radicals_Of_Shape(
               Result, Last, Size - 1 - I - J, J, I, Radicals);
         end loop;
      end loop;
   end if;
   Assert(Last = Length);
end Generate_Radicals_Of_Size;

---------------------------------------------------------------------

procedure Generate_Bond_Centered_Paraffins (
      Result   : in     Paraffin_Array_Pointer;
      Last     : in out Natural;
      Radicals : in     Radical_Array_Pointer   ) is
begin
   Assert(Result /= null);
   Assert(Result'First - 1 <= Last and Last <= Result'Last - 1);
   Assert(Radicals /= null);
   for I in 1 .. Radicals'Last loop
      for J in 1 .. I loop
         Last := Last + 1;
         Result(Last) := (Bond_Centered,
            (Radicals(I)'Access, Radicals(J)'Access, null, null));
      end loop;
   end loop;
end Generate_Bond_Centered_Paraffins;

---------------------------------------------------------------------

procedure Generate_Carbon_Centered_Paraffins (
      Result         : in     Paraffin_Array_Pointer;
      Last           : in out Natural;
      Size_1, Size_2,
      Size_3, Size_4 : in     Natural;
      Radicals       : in     Radical_Array_Pointers ) is
begin
   Assert(Result /= null);
   Assert(Result'First - 1 <= Last and Last <= Result'Last - 1);
```

```
   Assert(Size_1 <= Size_2 and Size_2 <= Size_3 and Size_3 <= Size_4);
   Assert(Radicals'First <= Size_4 and Size_1 <= Radicals'Last);
   for I in 1 .. Radicals(Size_1)'Last loop
      for J in 1 .. Last_Index(
            Size_1, Size_2, I, Radicals(Size_2)'Last) loop
         for K in 1 .. Last_Index(
               Size_2, Size_3, J, Radicals(Size_3)'Last) loop
            for L in 1 .. Last_Index(
                  Size_3, Size_4, K, Radicals(Size_4)'Last) loop
               Last := Last + 1;
               Result(Last) := (Carbon_Centered,
                  (Radicals(Size_1)(I)'Access,
                   Radicals(Size_2)(J)'Access,
                   Radicals(Size_3)(K)'Access,
                   Radicals(Size_4)(L)'Access ));
            end loop;
         end loop;
      end loop;
   end loop;
end Generate_Carbon_Centered_Paraffins;

--------------------------------------------------------------------

procedure Generate_Paraffins_Of_Size (
      Result   :    out Paraffin_Array_Pointer;
      Size     : in     Positive;
      Radicals : in     Radical_Array_Pointers ) is

   Length : Positive;
   Last   : Natural;

begin
   Assert(Radicals'First = 0 and Size/2 <= Radicals'Last);
   Length := Num_Paraffins(Size, Radicals);
   Result := new Paraffin_Array (1 .. Length);
   Last := 0;
   if Size mod 2 = 0 then
      Generate_Bond_Centered_Paraffins(
         Result, Last, Radicals(Size/2));
   end if;
   for I in (Size + 2)/4 .. (Size - 1)/2 loop
      for J in (Size + 1 - I)/3 .. Min(I, Size - 1 - I) loop
         for K in (Size - I - J)/2 .. Min(J, Size - 1 - I - J) loop
            Generate_Carbon_Centered_Paraffins(Result,
               Last, Size - 1 - I - J - K, K, J, I, Radicals);
         end loop;
      end loop;
   end loop;
   Assert(Last = Length);
end Generate_Paraffins_Of_Size;

--------------------------------------------------------------------

procedure Generate_Paraffins (
      Radicals  :    out Radical_Array_Pointers;
```

```
      Paraffins :    out Paraffin_Array_Pointers) is
   begin
      Assert(Radicals'First = 0 and Radicals'Last = Paraffins'Last/2);
      for R in Radicals'Range loop
         Generate_Radicals_Of_Size(
            Radicals(R), R, Radicals(0 .. R - 1));
      end loop;
      for P in Paraffins'Range loop
         Generate_Paraffins_Of_Size(
            Paraffins(P), P, Radicals(0 .. P/2));
      end loop;
   end Generate_Paraffins;

   ----------------------------------------------------------------------

end Paraffins_Problem;

-------------------------------------------------------------------------
```

B.2.2 Parallel Solution to the Paraffins Problem

Declaration of Parallel Solution to the Paraffins Problem

```
----------------------------------------------------------------------
-- Paraffins_Problem Package Declaration (Parallel)
----------------------------------------------------------------------

package Paraffins_Problem is

   type Radical_Kind is (Hydrogen, Carboniferous);
   type Radical;
   type Radical_Pointer is access all Radical;
   type Radical_Pointers is array (Integer range <>) of Radical_Pointer;
   type Radical is
      record
         Kind  : Radical_Kind;
         Bonds : Radical_Pointers (1 .. 3);
      end record;
   type Radical_Array is array (Integer range <>) of aliased Radical;
   type Radical_Array_Pointer is access Radical_Array;
   type Radical_Array_Pointers is
      array (Natural range <>) of Radical_Array_Pointer;

   type Paraffin_Kind is (Bond_Centered, Carbon_Centered);
   type Paraffin is
      record
         Kind  : Paraffin_Kind;
         Bonds : Radical_Pointers (1 .. 4);
      end record;
   type Paraffin_Array is array (Integer range <>) of Paraffin;
   type Paraffin_Array_Pointer is access Paraffin_Array;
   type Paraffin_Array_Pointers is
      array (Positive range <>) of Paraffin_Array_Pointer;

   procedure Deallocate (Radicals : in out Radical_Array_Pointers);

   procedure Deallocate (Paraffins : in out Paraffin_Array_Pointers);

   procedure Generate_Paraffins (
         Radicals  :    out Radical_Array_Pointers;
         Paraffins :    out Paraffin_Array_Pointers);
   --| Requires:
   --|    Radicals'First = 0 and Radicals'Last = Paraffins'Last/2.
   --| Ensures:
   --|    for all R in Radicals'Range :
   --|       All_Radicals_Of_Size(R, Radicals) and
   --|    for all P in Paraffins'Range :
   --|       All_Paraffins_Of_Size(P, Paraffins).

end Paraffins_Problem;

----------------------------------------------------------------------
```

Definition of Parallel Solution to the Paraffins Problem

```
------------------------------------------------------------------------
-- Paraffins_Problem Package Body (Parallel)
------------------------------------------------------------------------

with Assertions; use Assertions;
with Ada.Unchecked_Deallocation;

package body Paraffins_Problem is

   ---------------------------------------------------------------------

   procedure Deallocate is
      new Ada.Unchecked_Deallocation (
         Object => Radical_Array, Name => Radical_Array_Pointer);

   ---------------------------------------------------------------------

   procedure Deallocate is
      new Ada.Unchecked_Deallocation (
         Object => Paraffin_Array, Name => Paraffin_Array_Pointer);

   ---------------------------------------------------------------------

   procedure Deallocate (Radicals : in out Radical_Array_Pointers) is
   begin
      for R in Radicals'Range loop Deallocate(Radicals(R)); end loop;
   end Deallocate;

   ---------------------------------------------------------------------

   procedure Deallocate (Paraffins : in out Paraffin_Array_Pointers) is
   begin
      for P in Paraffins'Range loop Deallocate(Paraffins(P)); end loop;
   end Deallocate;

   ---------------------------------------------------------------------

   function Min (X, Y : Integer) return Integer renames Integer'Min;

   ---------------------------------------------------------------------

   type Bonded_Radical_Sizes is array (Positive range <>) of Natural;

   function Num_Arrangements (
         Radicals : Radical_Array_Pointers;
         Sizes    : Bonded_Radical_Sizes   ) return Positive is

      Length      : Positive;
      Numerator   : Positive;
      Denominator : Positive;
      Count       : Natural;

   begin
      Numerator := Radicals(Sizes(Sizes'First))'Length;
```

```
   Denominator := 1;
   Count := 0;
   for R in Sizes'First + 1 .. Sizes'Last loop
      Assert(Sizes(R - 1) <= Sizes(R));
      if Sizes(R) = Sizes(R - 1) then
         Count := Count + 1;
      else
         Count := 0;
      end if;
      Length := Radicals(Sizes(R))'Length;
      Numerator := Numerator*(Length + Count);
      Denominator := Denominator*(Count + 1);
   end loop;
   return Numerator/Denominator;
end Num_Arrangements;

--------------------------------------------------------------------

function Num_Radicals (
      Size     : Natural;
      Radicals : Radical_Array_Pointers) return Positive is

   Result : Natural;

begin
   Assert(Radicals'First = 0 and Size - 1 <= Radicals'Last);
   if Size = 0 then
      Result := 1;
   else
      Result := 0;
      for I in (Size + 1)/3 .. Size - 1 loop
         for J in (Size - I)/2 .. Min(I, Size - 1 - I) loop
            Result := Result + Num_Arrangements(
               Radicals, (1 => Size - 1 - I - J, 2 => J, 3 => I));
         end loop;
      end loop;
   end if;
   return Result;
end Num_Radicals;

--------------------------------------------------------------------

function Num_Paraffins (
      Size     : Positive;
      Radicals : Radical_Array_Pointers) return Positive is

   Result : Natural;

begin
   Assert(Radicals'First = 0 and Size/2 <= Radicals'Last);
   Result := 0;
   if Size mod 2 = 0 then
      Result := Result +
         Radicals(Size/2)'Length*(Radicals(Size/2)'Length + 1)/2;
   end if;
```

```
   for I in (Size + 2)/4 .. (Size - 1)/2 loop
      for J in (Size + 1 - I)/3 .. Min(I, Size - 1 - I) loop
         for K in (Size - I - J)/2 .. Min(J, Size - 1 - I - J) loop
            Result := Result + Num_Arrangements(Radicals,
               (1 => Size - 1 - I - J - K, 2 => K, 3 => J, 4 => I));
         end loop;
      end loop;
   end loop;
   return Result;
end Num_Paraffins;

---------------------------------------------------------------------

function Last_Index (
      Outer_Size, This_Size  : Natural;
      Outer_Index, This_Last : Integer ) return Integer is
begin
   if Outer_Size = This_Size then
      return Outer_Index;
   else
      return This_Last;
   end if;
end Last_Index;

---------------------------------------------------------------------

procedure Radicals_Of_Shape (
      Result                 : in     Radical_Array_Pointer;
      Last                   : in out Natural;
      Size_1, Size_2, Size_3 : in     Natural;
      Radicals               : in     Radical_Array_Pointers) is
begin
   Assert(Result /= null);
   Assert(Result'First - 1 <= Last and Last <= Result'Last - 1);
   Assert(Size_1 <= Size_2 and Size_2 <= Size_3);
   Assert(Radicals'First <= Size_3 and Size_1 <= Radicals'Last);
   for I in 1 .. Radicals(Size_1)'Last loop
      for J in 1 .. Last_Index(
            Size_1, Size_2, I, Radicals(Size_2)'Last) loop
         for K in 1 .. Last_Index(
               Size_2, Size_3, J, Radicals(Size_3)'Last) loop
            Last := Last + 1;
            Result(Last) := (Carboniferous,
               (Radicals(Size_1)(I)'Access,
                Radicals(Size_2)(J)'Access,
                Radicals(Size_3)(K)'Access ));
         end loop;
      end loop;
   end loop;
end Radicals_Of_Shape;

---------------------------------------------------------------------

procedure Generate_Radicals_Of_Size (
      Result   :    out Radical_Array_Pointer;
```

```
      Size     : in     Natural;
      Radicals : in     Radical_Array_Pointers) is

   Length : Positive;
   Last   : Natural;

begin
   Assert(Radicals'First = 0 and Size - 1 <= Radicals'Last);
   Length := Num_Radicals(Size, Radicals);
   Result := new Radical_Array (1 .. Length);
   Last := 0;
   if Size = 0 then
      Last := Last + 1;
      Result(Last) := (Hydrogen, (null, null, null));
   else
      for I in (Size + 1)/3 .. Size - 1 loop
         for J in (Size - I)/2 .. Min(I, Size - 1 - I) loop
            Radicals_Of_Shape(
               Result, Last, Size - 1 - I - J, J, I, Radicals);
         end loop;
      end loop;
   end if;
   Assert(Last = Length);
end Generate_Radicals_Of_Size;

---------------------------------------------------------------------

procedure Count_Bond_Centered (
      Size             : in     Positive;
      Radicals         : in     Radical_Array_Pointer;
      Num_Bond_Centered :    out Natural               ) is
begin
   Assert(Radicals /= null);
   if Size mod 2 /= 0 then
      Num_Bond_Centered := 0;
   else
      Num_Bond_Centered := Radicals'Length*(Radicals'Length + 1)/2;
   end if;
end Count_Bond_Centered;

---------------------------------------------------------------------

type Indices is array (Natural range <>,
                       Natural range <>,
                       Natural range <> ) of Positive;

procedure Count_Carbon_Centered (
      Size               : in     Positive;
      Radicals           : in     Radical_Array_Pointers;
      First              :    out Indices;
      Num_Carbon_Centered :    out Natural               ) is
begin
   Assert(Radicals'First = 0 and Size/2 <= Radicals'Last);
   Num_Carbon_Centered := 0;
   for I in (Size + 2)/4 .. (Size - 1)/2 loop
```

```
      for J in (Size + 1 - I)/3 .. Min(I, Size - 1 - I) loop
         for K in (Size - I - J)/2 .. Min(J, Size - 1 - I - J) loop
            First(I, J, K) := Num_Carbon_Centered + 1;
            Num_Carbon_Centered := Num_Carbon_Centered +
               Num_Arrangements(Radicals, (1 => Size - 1 - I - J - K,
                  2 => K, 3 => J, 4 => I));
         end loop;
      end loop;
   end loop;
end Count_Carbon_Centered;

---------------------------------------------------------------------

procedure Generate_Bond_Centered_Paraffins (
      Result      : in     Paraffin_Array_Pointer;
      Radicals    : in     Radical_Array_Pointer;
      Num_Threads : in     Positive               ) is
begin
   Assert(Result /= null);
   Assert(Radicals /= null);
   pragma Parallelizable_Loop(Cyclic, Num_Threads);
   for I in 1 .. Radicals'Last loop
      declare
         Base : constant Natural := (I*(I - 1))/2;
      begin
         for J in 1 .. I loop
            Result(Base + J) := (Bond_Centered,
               (Radicals(I)'Access, Radicals(J)'Access, null, null));
         end loop;
      end;
   end loop;
end Generate_Bond_Centered_Paraffins;

---------------------------------------------------------------------

procedure Generate_Carbon_Centered_Paraffins (
      Result         : in     Paraffin_Array_Pointer;
      First          : in     Positive;
      Size_1, Size_2,
      Size_3, Size_4 : in     Natural;
      Radicals       : in     Radical_Array_Pointers ) is

   Index : Positive;

begin
   Assert(Result /= null);
   Assert(Result'First <= First and First <= Result'Last);
   Assert(Size_1 <= Size_2 and Size_2 <= Size_3 and Size_3 <= Size_4);
   Assert(Radicals'First <= Size_4 and Size_1 <= Radicals'Last);
   Index := First;
   for I in 1 .. Radicals(Size_1)'Last loop
      for J in 1 .. Last_Index(
            Size_1, Size_2, I, Radicals(Size_2)'Last) loop
         for K in 1 .. Last_Index(
               Size_2, Size_3, J, Radicals(Size_3)'Last) loop
```

```
          for L in 1 .. Last_Index(
                Size_3, Size_4, K, Radicals(Size_4)'Last) loop
             Result(Index) := (Carbon_Centered,
                (Radicals(Size_1)(I)'Access,
                 Radicals(Size_2)(J)'Access,
                 Radicals(Size_3)(K)'Access,
                 Radicals(Size_4)(L)'Access ));
             Index := Index + 1;
          end loop;
       end loop;
    end loop;
 end loop;
end Generate_Carbon_Centered_Paraffins;

---------------------------------------------------------------------

procedure Generate_Paraffins_Of_Size (
      Result   :    out Paraffin_Array_Pointer;
      Size     : in     Positive;
      Radicals : in     Radical_Array_Pointers ) is

   Num_Bond_Centered   : Natural;
   Num_Carbon_Centered : Natural;
   Num_Paraffins       : Positive;
   First               : Indices ((Size + 2)/4 .. (Size - 1)/2,
                                  (Size + 3)/6 .. (Size - 1)/2,
                                  0 .. (Size - 1)/3);
begin
   Assert(Radicals'First = 0 and Size/2 <= Radicals'Last);
   Count_Bond_Centered(Size, Radicals(Size/2), Num_Bond_Centered);
   Count_Carbon_Centered(Size, Radicals, First, Num_Carbon_Centered);
   Num_Paraffins := Num_Bond_Centered + Num_Carbon_Centered;
   Result := new Paraffin_Array (1 .. Num_Paraffins);
   begin
      pragma Parallelizable_Sequence;
      if Size mod 2 = 0 then
         Generate_Bond_Centered_Paraffins(
            Result, Radicals(Size/2), Num_Threads => Size);
      end if;
      pragma Parallelizable_Loop;
      for I in (Size + 2)/4 .. (Size - 1)/2 loop
         pragma Parallelizable_Loop;
         for J in (Size + 1 - I)/3 .. Min(I, Size - 1 - I) loop
            for K in (Size - I - J)/2 ..
                     Min(J, Size - 1 - I - J) loop
               Generate_Carbon_Centered_Paraffins(
                  Result, Num_Bond_Centered + First(I, J, K),
                  Size - 1 - I - J - K, K, J, I, Radicals);
            end loop;
         end loop;
      end loop;
   end;
end Generate_Paraffins_Of_Size;

---------------------------------------------------------------------
```

```
   procedure Generate_Paraffins (
         Radicals  :    out Radical_Array_Pointers;
         Paraffins :    out Paraffin_Array_Pointers) is
   begin
      Assert(Radicals'First = 0 and Radicals'Last = Paraffins'Last/2);
      for R in Radicals'Range loop
         Generate_Radicals_Of_Size(
            Radicals(R), R, Radicals(0 .. R - 1));
      end loop;
      pragma Parallelizable_Loop(On_Demand, Num_Threads => 2);
      for P in reverse Paraffins'Range loop
         Generate_Paraffins_Of_Size(
            Paraffins(P), P, Radicals(0 .. P/2));
      end loop;
   end Generate_Paraffins;

   ---------------------------------------------------------------------

end Paraffins_Problem;

------------------------------------------------------------------------
```

B.3 Mergesort of a Linked List

B.3.1 Linked Lists with Mutable Links

Declaration of Linked Lists with Mutable Links

```
-----------------------------------------------------------------------
-- Lists Generic Package Declaration (Mutable Links)
-----------------------------------------------------------------------

with Ada.Finalization; use Ada.Finalization;
with Ada.Unchecked_Deallocation;

generic
   type Element is private;
package Lists is

   --------------------------------------------------------------------

   List_Error : exception;

   --------------------------------------------------------------------

   type Elements is array (Integer range <>) of Element;

   type Block (Block_Length : Positive) is
      record
         Data   : Elements (1 .. Block_Length);
         Length : Positive;
      end record;
   type Block_Access is access Block;

   procedure Deallocate is new
      Ada.Unchecked_Deallocation(
         Object => Block, Name => Block_Access);

   --------------------------------------------------------------------

   type List (Block_Length : Positive) is
      new Limited_Controlled with private;

   --------------------------------------------------------------------

   procedure Initialize (L : in out List);

   procedure Finalize (L : in out List);

   procedure Put (
         L    : in out List;
         Item : in     Block_Access);

   function End_Of_List (L : List) return Boolean;

   procedure Get (
         L    : in out List;
```

```
      Item : out    Block_Access);

   procedure Get (
      L        : in out List;
      Item     : out    Block_Access;
      Past_End : out    Boolean      );

   ---------------------------------------------------------------------

private

   type Node;
   type Pointer is access Node;
   type Node is
      record
         Item : Block_Access;
         Next : Pointer;
      end record;
   type List (Block_Length : Positive) is new Limited_Controlled with
      record
         Head : Pointer;
         Tail : Pointer;
      end record;

end Lists;

------------------------------------------------------------------------
```

Definition of Linked Lists with Mutable Links

```
-----------------------------------------------------------------------
-- Lists Generic Package Body (Mutable Links)
-----------------------------------------------------------------------

with Ada.Unchecked_Deallocation;

package body Lists is

   --------------------------------------------------------------------

   procedure Deallocate is
      new Ada.Unchecked_Deallocation(Object => Node, Name => Pointer);

   --------------------------------------------------------------------

   procedure Initialize (L : in out List) is
   begin
      L.Head := new Node;
      L.Head.Item := null;
      L.Head.Next := null;
      L.Tail := L.Head;
   end Initialize;

   --------------------------------------------------------------------

   procedure Finalize (L : in out List) is

      P, Next : Pointer;

   begin
      P := L.Head;
      while P /= null loop
         Next := P.Next;
         Deallocate(P);
         P := Next;
      end loop;
   end Finalize;

   --------------------------------------------------------------------

   procedure Put (
         L    : in out List;
         Item : in     Block_Access) is

      New_Tail : Pointer;

   begin
      New_Tail := new Node;
      New_Tail.Item := Item;
      New_Tail.Next := null;
      L.Tail.Next := New_Tail;
      L.Tail := New_Tail;
   end Put;
```

```
   ---------------------------------------------------------------------

   function End_Of_List (L : List) return Boolean is
   begin
      return L.Head.Next = null;
   end End_Of_List;

   ---------------------------------------------------------------------

   procedure Get (
         L    : in out List;
         Item :    out Block_Access) is

      Old_Head : Pointer;

   begin
      if L.Head.Next = null then
         raise List_Error;
      else
         Old_Head := L.Head;
         L.Head := L.Head.Next;
         Item := L.Head.Item;
         Deallocate(Old_Head);
      end if;
   end Get;

   ---------------------------------------------------------------------

   procedure Get (
         L        : in out List;
         Item     :    out Block_Access;
         Past_End :    out Boolean      ) is

      Old_Head : Pointer;

   begin
      if L.Head.Next = null then
         Past_End := True;
      else
         Past_End := False;
         Old_Head := L.Head;
         L.Head := L.Head.Next;
         Item := L.Head.Item;
         Deallocate(Old_Head);
      end if;
   end Get;

   ---------------------------------------------------------------------

end Lists;

------------------------------------------------------------------------
```

B.3.2 Sequential Mergesort of a Linked List with Mutable Links

Declaration of Sequential Mergesort of a Linked List with Mutable Links

```
------------------------------------------------------------------------
-- Generic Sorting Package Declaration (Sequential/Mutable Links)
------------------------------------------------------------------------

with Lists;

generic
   type Element is private;
   with function "<" (Left, Right : Element) return Boolean is <>;
   with function "<=" (Left, Right : Element) return Boolean is <>;
   with function ">" (Left, Right : Element) return Boolean is <>;
   with function ">=" (Left, Right : Element) return Boolean is <>;
package Sorting is

   package Element_Lists is new Lists (Element);
   use Element_Lists;

   procedure Mergesort (Unsorted, Sorted : in out List);
   --| Requires:
   --|    Empty(Sorted).
   --| Ensures:
   --|    Ascending(Sorted) and Permutation(Sorted, in Unsorted).

end Sorting;

------------------------------------------------------------------------
```

Definition of Sequential Mergesort of a Linked List with Mutable Links

```
----------------------------------------------------------------------
-- Generic Sorting Package Body (Sequential/Mutable Links)
----------------------------------------------------------------------

with Assertions; use Assertions;

package body Sorting is

   -------------------------------------------------------------------

   procedure Insertion_Sort (Data : in out Elements) is
   begin
      for I in Data'First .. Data'Last - 1 loop
         declare
            Temp : Element;
            Pos  : Integer range Data'Range;
         begin
            Temp := Data(I + 1);
            Pos  := Data'First;
            for J in reverse Data'First .. I loop
               if Temp < Data(J) then
                  Data(J + 1) := Data(J);
               else
                  Pos := J + 1;
                  exit;
               end if;
            end loop;
            Data(Pos) := Temp;
         end;
      end loop;
   end Insertion_Sort;

   -------------------------------------------------------------------

   procedure Swap (X, Y : in out Element) is

      Temp : Element;

   begin
      Temp := X;
      X := Y;
      Y := Temp;
   end Swap;

   -------------------------------------------------------------------

   procedure Partition (
         Data        : in out Elements;
         Pivot_Index : out Integer     ) is

      First  : constant Integer := Data'First;
      Last   : constant Integer := Data'Last;
      Length : constant Integer := Last - First + 1;
      Pivot_Value : Element;
```

```
   Left, Right : Integer range Data'Range;

begin
   Assert(Length >= 3);
   Swap(Data(First + 1), Data((First + Last)/2));
   if Data(First + 1) > Data(Last) then
      Swap(Data(First + 1), Data(Last));
   end if;
   if Data(First) > Data(Last) then
      Swap(Data(First), Data(Last));
   end if;
   if Data(First + 1) > Data(First) then
      Swap(Data(First + 1), Data(First));
   end if;
   Pivot_Value := Data(First);
   Left := First + 1; Right := Last;
   loop
      Left := Left + 1;
      while Data(Left) < Pivot_Value loop
         Left := Left + 1;
      end loop;
      Right := Right - 1;
      while Data(Right) > Pivot_Value loop
         Right := Right - 1;
      end loop;
      exit when Right < Left;
      Swap(Data(Left), Data(Right));
   end loop;
   Data(First) := Data(Right);
   Data(Right) := Pivot_Value;
   Pivot_Index := Right;
   Assert(First <= Pivot_Index and Pivot_Index <= Last);
end Partition;

----------------------------------------------------------------------

Base_Length : constant Positive := 16;

----------------------------------------------------------------------

procedure Quicksort (Data : in out Elements) is

   First  : constant Integer := Data'First;
   Last   : constant Integer := Data'Last;
   Length : constant Integer := Last - First + 1;

begin
   Assert(2 <= Base_Length);
   if Length <= Base_Length then
      Insertion_Sort(Data);
   else
      declare
         Pivot_Index : Integer range Data'Range;
      begin
         Partition(Data, Pivot_Index);
```

```
         Quicksort(Data(First .. Pivot_Index - 1));
         Quicksort(Data(Pivot_Index + 1 .. Last));
      end;
   end if;
end Quicksort;

---------------------------------------------------------------------

procedure Split (Input, Left, Right : in out List) is

   Block    : Block_Access;
   Finished : Boolean;

begin
   Get(Input, Block, Finished);
   while not Finished loop
      Put(Left, Block);
      Get(Input, Block, Finished);
      if not Finished then
         Put(Right, Block);
         Get(Input, Block, Finished);
      end if;
   end loop;
end Split;

---------------------------------------------------------------------

procedure Merge (Left, Right, Output : in out List) is

   Left_Finished, Right_Finished : Boolean;
   Left_Block, Right_Block, Output_Block : Block_Access;
   L, R, O : Natural;

begin
   Assert(not (End_Of_List(Left) and End_Of_List(Right)));
   Get(Left, Left_Block, Left_Finished);
   Get(Right, Right_Block, Right_Finished);
   Output_Block := new Block (Output.Block_length);
   L := 1; R := 1; O := 1;
   Assert(not (Left_Finished and Right_Finished));
   if not (Left_Finished or Right_Finished) then
      loop
         if O > Output.Block_Length then
            Output_Block.Length := Output.Block_Length;
            Put(Output, Output_Block);
            Output_Block := new Block (Output.Block_length);
            O := 1;
         end if;
         if Left_Block.Data(L) <= Right_Block.Data(R) then
            Output_Block.Data(O) := Left_Block.Data(L);
            O := O + 1; L := L + 1;
            if L > Left_Block.Length then
               Deallocate(Left_Block);
               Get(Left, Left_Block, Left_Finished);
               exit when Left_Finished;
```

```
            L := 1;
         end if;
      else
         Output_Block.Data(O) := Right_Block.Data(R);
         O := O + 1; R := R + 1;
         if R > Right_Block.Length then
            Deallocate(Right_Block);
            Get(Right, Right_Block, Right_Finished);
            exit when Right_Finished;
            R := 1;
         end if;
      end if;
    end loop;
  end if;
  if not Left_Finished then
    loop
      if O > Output.Block_Length then
         Output_Block.Length := Output.Block_Length;
         Put(Output, Output_Block);
         Output_Block := new Block (Output.Block_length);
         O := 1;
      end if;
      Output_Block.Data(O) := Left_Block.Data(L);
      O := O + 1; L := L + 1;
      if L > Left_Block.Length then
         Deallocate(Left_Block);
         Get(Left, Left_Block, Left_Finished);
         exit when Left_Finished;
         L := 1;
      end if;
    end loop;
  else
    loop
      if O > Output.Block_Length then
         Output_Block.Length := Output.Block_Length;
         Put(Output, Output_Block);
         Output_Block := new Block (Output.Block_length);
         O := 1;
      end if;
      Output_Block.Data(O) := Right_Block.Data(R);
      O := O + 1; R := R + 1;
      if R > Right_Block.Length then
         Deallocate(Right_Block);
         Get(Right, Right_Block, Right_Finished);
         exit when Right_Finished;
         R := 1;
      end if;
    end loop;
  end if;
  Output_Block.Length := O - 1;
  Put(Output, Output_Block);
end Merge;

-------------------------------------------------------------------
```

```
   procedure Mergesort (Unsorted, Sorted : in out List) is

      Block_1, Block_2    : Block_Access;
      Get_Past_End        : Boolean;
      Left, Sorted_Left   : List (Unsorted.Block_Length);
      Right, Sorted_Right : List (Unsorted.Block_Length);

   begin
      Get(Unsorted, Block_1, Get_Past_End);
      if not Get_Past_End then
         Get(Unsorted, Block_2, Get_Past_End);
         if Get_Past_End then
            Quicksort(Block_1.Data(1 .. Block_1.Length));
            Put(Sorted, Block_1);
         else
            Put(Left, Block_1); Put(Right, Block_2);
            Split(Unsorted, Left, Right);
            Mergesort(Left, Sorted_Left);
            Mergesort(Right, Sorted_Right);
            Merge(Sorted_Left, Sorted_Right, Sorted);
         end if;
      end if;
   end Mergesort;

   ---------------------------------------------------------------------

end Sorting;

------------------------------------------------------------------------
```

B.3.3 Parallel Mergesort of a Linked List with Mutable Links

Declaration of Parallel Mergesort of a Linked List with Mutable Links

```
------------------------------------------------------------------------
-- Generic Sorting Package Declaration (Parallel/Mutable Links)
------------------------------------------------------------------------

with Lists;

generic
   type Element is private;
   with function "<" (Left, Right : Element) return Boolean is <>;
   with function "<=" (Left, Right : Element) return Boolean is <>;
   with function ">" (Left, Right : Element) return Boolean is <>;
   with function ">=" (Left, Right : Element) return Boolean is <>;
package Sorting is

   package Element_Lists is new Lists (Element);
   use Element_Lists;

   procedure Mergesort (Unsorted, Sorted : in out List;
                        Parallel_Depth   : in     Natural);
   --| Requires:
   --|    Empty(Sorted).
   --| Ensures:
   --|    Ascending(Sorted) and Permutation(Sorted, in Unsorted).

end Sorting;

------------------------------------------------------------------------
```

Definition of Parallel Mergesort of a Linked List with Mutable Links

```
-----------------------------------------------------------------------
-- Generic Sorting Package Body (Parallel/Mutable Links)
-----------------------------------------------------------------------

with Assertions; use Assertions;

package body Sorting is

   --------------------------------------------------------------------

   procedure Insertion_Sort (Data : in out Elements) is
   begin
      for I in Data'First .. Data'Last - 1 loop
         declare
            Temp : Element;
            Pos  : Integer range Data'Range;
         begin
            Temp := Data(I + 1);
            Pos  := Data'First;
            for J in reverse Data'First .. I loop
               if Temp < Data(J) then
                  Data(J + 1) := Data(J);
               else
                  Pos := J + 1;
                  exit;
               end if;
            end loop;
            Data(Pos) := Temp;
         end;
      end loop;
   end Insertion_Sort;

   --------------------------------------------------------------------

   procedure Swap (X, Y : in out Element) is

      Temp : Element;

   begin
      Temp := X;
      X := Y;
      Y := Temp;
   end Swap;

   --------------------------------------------------------------------

   procedure Partition (
         Data        : in out Elements;
         Pivot_Index : out Integer     ) is

      First  : constant Integer := Data'First;
      Last   : constant Integer := Data'Last;
      Length : constant Integer := Last - First + 1;
      Pivot_Value : Element;
```

```
   Left, Right : Integer range Data'Range;

begin
   Assert(Length >= 3);
   Swap(Data(First + 1), Data((First + Last)/2));
   if Data(First + 1) > Data(Last) then
      Swap(Data(First + 1), Data(Last));
   end if;
   if Data(First) > Data(Last) then
      Swap(Data(First), Data(Last));
   end if;
   if Data(First + 1) > Data(First) then
      Swap(Data(First + 1), Data(First));
   end if;
   Pivot_Value := Data(First);
   Left := First + 1; Right := Last;
   loop
      Left := Left + 1;
      while Data(Left) < Pivot_Value loop
         Left := Left + 1;
      end loop;
      Right := Right - 1;
      while Data(Right) > Pivot_Value loop
         Right := Right - 1;
      end loop;
      exit when Right < Left;
      Swap(Data(Left), Data(Right));
   end loop;
   Data(First) := Data(Right);
   Data(Right) := Pivot_Value;
   Pivot_Index := Right;
   Assert(First <= Pivot_Index and Pivot_Index <= Last);
end Partition;

---------------------------------------------------------------------

Base_Length : constant Positive := 16;

---------------------------------------------------------------------

procedure Quicksort (Data : in out Elements) is

   First  : constant Integer := Data'First;
   Last   : constant Integer := Data'Last;
   Length : constant Integer := Last - First + 1;

begin
   Assert(2 <= Base_Length);
   if Length <= Base_Length then
      Insertion_Sort(Data);
   else
      declare
         Pivot_Index : Integer range Data'Range;
      begin
         Partition(Data, Pivot_Index);
```

```
         Quicksort(Data(First .. Pivot_Index - 1));
         Quicksort(Data(Pivot_Index + 1 .. Last));
      end;
   end if;
end Quicksort;

----------------------------------------------------------------------

procedure Split (Input, Left, Right : in out List) is

   Block    : Block_Access;
   Finished : Boolean;

begin
   Get(Input, Block, Finished);
   while not Finished loop
      Put(Left, Block);
      Get(Input, Block, Finished);
      if not Finished then
         Put(Right, Block);
         Get(Input, Block, Finished);
      end if;
   end loop;
end Split;

----------------------------------------------------------------------

procedure Merge (Left, Right, Output : in out List) is

   Left_Finished, Right_Finished : Boolean;
   Left_Block, Right_Block, Output_Block : Block_Access;
   L, R, O : Natural;

begin
   Assert(not (End_Of_List(Left) and End_Of_List(Right)));
   Get(Left, Left_Block, Left_Finished);
   Get(Right, Right_Block, Right_Finished);
   Output_Block := new Block (Output.Block_length);
   L := 1; R := 1; O := 1;
   Assert(not (Left_Finished and Right_Finished));
   if not (Left_Finished or Right_Finished) then
      loop
         if O > Output.Block_Length then
            Output_Block.Length := Output.Block_Length;
            Put(Output, Output_Block);
            Output_Block := new Block (Output.Block_length);
            O := 1;
         end if;
         if Left_Block.Data(L) <= Right_Block.Data(R) then
            Output_Block.Data(O) := Left_Block.Data(L);
            O := O + 1; L := L + 1;
            if L > Left_Block.Length then
               Deallocate(Left_Block);
               Get(Left, Left_Block, Left_Finished);
               exit when Left_Finished;
```

```
            L := 1;
         end if;
      else
         Output_Block.Data(O) := Right_Block.Data(R);
         O := O + 1; R := R + 1;
         if R > Right_Block.Length then
            Deallocate(Right_Block);
            Get(Right, Right_Block, Right_Finished);
            exit when Right_Finished;
            R := 1;
         end if;
      end if;
   end loop;
end if;
if not Left_Finished then
   loop
      if O > Output.Block_Length then
         Output_Block.Length := Output.Block_Length;
         Put(Output, Output_Block);
         Output_Block := new Block (Output.Block_length);
         O := 1;
      end if;
      Output_Block.Data(O) := Left_Block.Data(L);
      O := O + 1; L := L + 1;
      if L > Left_Block.Length then
         Deallocate(Left_Block);
         Get(Left, Left_Block, Left_Finished);
         exit when Left_Finished;
         L := 1;
      end if;
   end loop;
else
   loop
      if O > Output.Block_Length then
         Output_Block.Length := Output.Block_Length;
         Put(Output, Output_Block);
         Output_Block := new Block (Output.Block_length);
         O := 1;
      end if;
      Output_Block.Data(O) := Right_Block.Data(R);
      O := O + 1; R := R + 1;
      if R > Right_Block.Length then
         Deallocate(Right_Block);
         Get(Right, Right_Block, Right_Finished);
         exit when Right_Finished;
         R := 1;
      end if;
   end loop;
end if;
Output_Block.Length := O - 1;
Put(Output, Output_Block);
end Merge;

---------------------------------------------------------------------
```

```
   procedure Mergesort (Unsorted, Sorted : in out List;
                        Parallel_Depth   : in     Natural) is

      Block_1, Block_2    : Block_Access;
      Get_Past_End        : Boolean;
      Left, Sorted_Left   : List (Unsorted.Block_Length);
      Right, Sorted_Right : List (Unsorted.Block_Length);

   begin
      Get(Unsorted, Block_1, Get_Past_End);
      if not Get_Past_End then
         Get(Unsorted, Block_2, Get_Past_End);
         if Get_Past_End then
            Quicksort(Block_1.Data(1 .. Block_1.Length));
            Put(Sorted, Block_1);
         else
            Put(Left, Block_1); Put(Right, Block_2);
            Split(Unsorted, Left, Right);
            if Parallel_Depth = 0 then
               Mergesort(Left, Sorted_Left, 0);
               Mergesort(Right, Sorted_Right, 0);
            else
               pragma Parallelizable_Sequence;
               Mergesort(Left, Sorted_Left, Parallel_Depth - 1);
               Mergesort(Right, Sorted_Right, Parallel_Depth - 1);
            end if;
            Merge(Sorted_Left, Sorted_Right, Sorted);
         end if;
      end if;
   end Mergesort;

   ---------------------------------------------------------------------

end Sorting;

------------------------------------------------------------------------
```

B.3.4 Linked Lists with Single-Assignment Links

Declaration of Linked Lists with Single-Assignment Links

```
----------------------------------------------------------------------
-- Lists Generic Package Declaration (Single-Assignment Links)
----------------------------------------------------------------------

with Ada.Finalization; use Ada.Finalization;
with Ada.Unchecked_Deallocation;

generic
   type Element is private;
package Lists is

   -------------------------------------------------------------------

   List_Error : exception;

   -------------------------------------------------------------------

   type Elements is array (Integer range <>) of Element;

   type Block (Block_Length : Positive) is
      record
         Data   : Elements (1 .. Block_Length);
         Length : Positive;
      end record;
   type Block_Access is access Block;

   procedure Deallocate is new
      Ada.Unchecked_Deallocation(
         Object => Block, Name => Block_Access);

   -------------------------------------------------------------------

   type List (Block_Length : Positive) is
      new Limited_Controlled with private;

   -------------------------------------------------------------------

   procedure Initialize (L : in out List);

   procedure Finalize (L : in out List);

   procedure Put (
         L    : in out List;
         Item : in     Block_Access);

   procedure Close (L : in out List);

   function End_Of_List (L : List) return Boolean;

   procedure Get (
         L    : in out List;
```

```
      Item : out    Block_Access);

   procedure Get (
      L        : in out List;
      Item     : out    Block_Access;
      Past_End : out    Boolean      );

   ----------------------------------------------------------------------

private

   type Node;
   type Pointer is access Node;
   type Single_Pointer is new Pointer;
   pragma Single_Assignment(Single_Pointer);
   type Node is
      record
         Item : Block_Access;
         Next : Single_Pointer;
      end record;
   type List (Block_Length : Positive) is new Limited_Controlled with
      record
         Closed : Boolean;
         Head   : Pointer;
         Tail   : Pointer;
    end record;

end Lists;

-------------------------------------------------------------------------
```

Definition of Linked Lists with Single-Assignment Links

```
------------------------------------------------------------------------
-- Lists Generic Package Body (Single-Assignment Links)
------------------------------------------------------------------------

with Ada.Unchecked_Deallocation;

package body Lists is

   ---------------------------------------------------------------------

   procedure Deallocate is
      new Ada.Unchecked_Deallocation(Object => Node, Name => Pointer);

   ---------------------------------------------------------------------

   procedure Initialize (L : in out List) is
   begin
      L.Closed := False;
      L.Head := new Node;
      L.Head.Item := null;
      L.Tail := L.Head;
   end Initialize;

   ---------------------------------------------------------------------

   procedure Finalize (L : in out List) is

      P, Next : Pointer;

   begin
      P := L.Head;
      while P /= L.Tail loop
         Next := Pointer(P.Next);
         Deallocate(P);
         P := Next;
      end loop;
      Deallocate(P);
   end Finalize;

   ---------------------------------------------------------------------

   procedure Put (
         L    : in out List;
         Item : in     Block_Access) is

      New_Tail : Pointer;

   begin
      if L.Closed then
         raise List_Error;
      else
         New_Tail := new Node;
         New_Tail.Item := Item;
         L.Tail.Next := Single_Pointer(New_Tail);
```

```
      L.Tail := New_Tail;
   end if;
end Put;

----------------------------------------------------------------------

procedure Close (L : in out List) is
begin
   if L.Closed then
      raise List_Error;
   else
      L.Tail.Next := null;
      L.Closed := True;
   end if;
end Close;

----------------------------------------------------------------------

function End_Of_List (L : List) return Boolean is
begin
   return L.Head.Next = null;
end End_Of_List;

----------------------------------------------------------------------

procedure Get (
      L    : in out List;
      Item :    out Block_Access) is

   Old_Head, Head_Next : Pointer;

begin
   Head_Next := Pointer(L.Head.Next);
   if Head_Next = null then
      raise List_Error;
   else
      Old_Head := L.Head;
      L.Head := Head_Next;
      Item := L.Head.Item;
      Deallocate(Old_Head);
   end if;
end Get;

----------------------------------------------------------------------

procedure Get (
      L        : in out List;
      Item     :    out Block_Access;
      Past_End :    out Boolean      ) is

    Old_Head, Head_Next : Pointer;

begin
   Head_Next := Pointer(L.Head.Next);
   if Head_Next = null then
```

```
         Past_End := True;
      else
         Past_End := False;
         Old_Head := L.Head;
         L.Head := Head_Next;
         Item := L.Head.Item;
         Deallocate(Old_Head);
      end if;
   end Get;

   ---------------------------------------------------------------------

end Lists;

------------------------------------------------------------------------
```

B.3.5 Parallel Mergesort of a Linked List with Single-Assignment Links

Declaration of Parallel Mergesort of a Linked List with Single-Assignment Links

```
-------------------------------------------------------------------------
-- Generic Sorting Package Declaration (Parallel/Single-Assignment Links)
-------------------------------------------------------------------------

with Lists;

generic
   type Element is private;
   with function "<" (Left, Right : Element) return Boolean is <>;
   with function "<=" (Left, Right : Element) return Boolean is <>;
   with function ">" (Left, Right : Element) return Boolean is <>;
   with function ">=" (Left, Right : Element) return Boolean is <>;
package Sorting is

   package Element_Lists is new Lists (Element);
   use Element_Lists;

   procedure Mergesort (Unsorted, Sorted : in out List;
                        Parallel_Depth   : in     Natural);
   --| Requires:
   --|    Closed(Unsorted) and not Closed(Sorted) and Empty(Sorted).
   --| Ensures:
   --|    Closed(Sorted) and
   --|    Ascending(Sorted) and Permutation(Sorted, in Unsorted).

end Sorting;

-------------------------------------------------------------------------
```

Definition of Parallel Mergesort of a Linked List with Single-Assignment Links

```
----------------------------------------------------------------------
-- Generic Sorting Package Body (Parallel/Single-Assignment Links)
----------------------------------------------------------------------

with Assertions; use Assertions;
with System; use System;

package body Sorting is

   -------------------------------------------------------------------

   procedure Insertion_Sort (Data : in out Elements) is
   begin
      for I in Data'First .. Data'Last - 1 loop
         declare
            Temp : Element;
            Pos  : Integer range Data'Range;
         begin
            Temp := Data(I + 1);
            Pos  := Data'First;
            for J in reverse Data'First .. I loop
               if Temp < Data(J) then
                  Data(J + 1) := Data(J);
               else
                  Pos := J + 1;
                  exit;
               end if;
            end loop;
            Data(Pos) := Temp;
         end;
      end loop;
   end Insertion_Sort;

   -------------------------------------------------------------------

   procedure Swap (X, Y : in out Element) is

      Temp : Element;

   begin
      Temp := X;
      X := Y;
      Y := Temp;
   end Swap;

   -------------------------------------------------------------------

   procedure Partition (
         Data        : in out Elements;
         Pivot_Index : out Integer     ) is

      First  : constant Integer := Data'First;
      Last   : constant Integer := Data'Last;
      Length : constant Integer := Last - First + 1;
```

```
   Pivot_Value : Element;
   Left, Right : Integer range Data'Range;

begin
   Assert(Length >= 3);
   Swap(Data(First + 1), Data((First + Last)/2));
   if Data(First + 1) > Data(Last) then
      Swap(Data(First + 1), Data(Last));
   end if;
   if Data(First) > Data(Last) then
      Swap(Data(First), Data(Last));
   end if;
   if Data(First + 1) > Data(First) then
      Swap(Data(First + 1), Data(First));
   end if;
   Pivot_Value := Data(First);
   Left := First + 1; Right := Last;
   loop
      Left := Left + 1;
      while Data(Left) < Pivot_Value loop
         Left := Left + 1;
      end loop;
      Right := Right - 1;
      while Data(Right) > Pivot_Value loop
         Right := Right - 1;
      end loop;
      exit when Right < Left;
      Swap(Data(Left), Data(Right));
   end loop;
   Data(First) := Data(Right);
   Data(Right) := Pivot_Value;
   Pivot_Index := Right;
   Assert(First <= Pivot_Index and Pivot_Index <= Last);
end Partition;

---------------------------------------------------------------------

Base_Length : constant Positive := 16;

---------------------------------------------------------------------

procedure Quicksort (Data : in out Elements) is

   First  : constant Integer := Data'First;
   Last   : constant Integer := Data'Last;
   Length : constant Integer := Last - First + 1;

begin
   Assert(2 <= Base_Length);
   if Length <= Base_Length then
      Insertion_Sort(Data);
   else
      declare
         Pivot_Index : Integer range Data'Range;
      begin
```

```
         Partition(Data, Pivot_Index);
         Quicksort(Data(First .. Pivot_Index - 1));
         Quicksort(Data(Pivot_Index + 1 .. Last));
      end;
   end if;
end Quicksort;

---------------------------------------------------------------------

procedure Split (Input, Left, Right : in out List) is

   Block    : Block_Access;
   Finished : Boolean;

begin
   Get(Input, Block, Finished);
   while not Finished loop
      Put(Left, Block);
      Get(Input, Block, Finished);
      if not Finished then
         Put(Right, Block);
         Get(Input, Block, Finished);
      end if;
   end loop;
   Close(Left); Close(Right);
end Split;

---------------------------------------------------------------------

procedure Merge (Left, Right, Output : in out List) is

   Left_Finished, Right_Finished : Boolean;
   Left_Block, Right_Block, Output_Block : Block_Access;
   L, R, O : Natural;

begin
   Assert(not (End_Of_List(Left) and End_Of_List(Right)));
   Get(Left, Left_Block, Left_Finished);
   Get(Right, Right_Block, Right_Finished);
   Output_Block := new Block (Output.Block_length);
   L := 1; R := 1; O := 1;
   Assert(not (Left_Finished and Right_Finished));
   if not (Left_Finished or Right_Finished) then
      loop
         if O > Output.Block_Length then
            Output_Block.Length := Output.Block_Length;
            Put(Output, Output_Block);
            Output_Block := new Block (Output.Block_length);
            O := 1;
         end if;
         if Left_Block.Data(L) <= Right_Block.Data(R) then
            Output_Block.Data(O) := Left_Block.Data(L);
            O := O + 1; L := L + 1;
            if L > Left_Block.Length then
               Deallocate(Left_Block);
```

```
            Get(Left, Left_Block, Left_Finished);
            exit when Left_Finished;
            L := 1;
         end if;
      else
         Output_Block.Data(O) := Right_Block.Data(R);
         O := O + 1; R := R + 1;
         if R > Right_Block.Length then
            Deallocate(Right_Block);
            Get(Right, Right_Block, Right_Finished);
            exit when Right_Finished;
            R := 1;
         end if;
      end if;
   end loop;
end if;
if not Left_Finished then
   loop
      if O > Output.Block_Length then
         Output_Block.Length := Output.Block_Length;
         Put(Output, Output_Block);
         Output_Block := new Block (Output.Block_length);
         O := 1;
      end if;
      Output_Block.Data(O) := Left_Block.Data(L);
      O := O + 1; L := L + 1;
      if L > Left_Block.Length then
         Deallocate(Left_Block);
         Get(Left, Left_Block, Left_Finished);
         exit when Left_Finished;
         L := 1;
      end if;
   end loop;
else
   loop
      if O > Output.Block_Length then
         Output_Block.Length := Output.Block_Length;
         Put(Output, Output_Block);
         Output_Block := new Block (Output.Block_length);
         O := 1;
      end if;
      Output_Block.Data(O) := Right_Block.Data(R);
      O := O + 1; R := R + 1;
      if R > Right_Block.Length then
         Deallocate(Right_Block);
         Get(Right, Right_Block, Right_Finished);
         exit when Right_Finished;
         R := 1;
      end if;
   end loop;
end if;
Output_Block.Length := O - 1;
Put(Output, Output_Block);
Close(Output);
end Merge;
```

```
   ---------------------------------------------------------------------

   procedure Mergesort (Unsorted, Sorted : in out List;
                        Parallel_Depth   : in     Natural) is

      Block_1, Block_2    : Block_Access;
      Get_Past_End        : Boolean;
      Left, Sorted_Left   : List (Unsorted.Block_Length);
      Right, Sorted_Right : List (Unsorted.Block_Length);

   begin
      Get(Unsorted, Block_1, Get_Past_End);
      if Get_Past_End then
         Close(Sorted);
      else
         Get(Unsorted, Block_2, Get_Past_End);
         if Get_Past_End then
            Quicksort(Block_1.Data(1 .. Block_1.Length));
            Put(Sorted, Block_1);
            Close(Sorted);
         else
            Put(Left, Block_1); Put(Right, Block_2);
            Split(Unsorted, Left, Right);
            if Parallel_Depth = 0 then
               Mergesort(Left, Sorted_Left, 0);
               Mergesort(Right, Sorted_Right, 0);
               Merge(Sorted_Left, Sorted_Right, Sorted);
            else
               pragma Parallelizable_Sequence;
               Mergesort(Left, Sorted_Left, Parallel_Depth - 1);
               Mergesort(Right, Sorted_Right, Parallel_Depth - 1);
               pragma Priority(Default_Priority + Parallel_Depth);
               Merge(Sorted_Left, Sorted_Right, Sorted);
            end if;
         end if;
      end if;
   end Mergesort;

   ---------------------------------------------------------------------

end Sorting;

------------------------------------------------------------------------
```

B.4 LU Factorization

B.4.1 Sequential LU Factorization

Declaration of Sequential LU Factorization

```
------------------------------------------------------------------------
-- LU_Factorization Package Declaration (Sequential)
------------------------------------------------------------------------

package LU_Factorization is

   type Matrix is array (Integer range <>,
                         Integer range <> ) of Float;

   procedure LU_Factorize (A : in Matrix; LU : out Matrix);
   --| Requires:
   --|    A'Length > 0 and Square(A) and
   --|    Nonsingular(A) and Same_Bounds(LU, A).
   --| Ensures:
   --|    Unit_Lower_Triangle(LU)*Upper_Triangle(LU) = A.

end LU_Factorization;

------------------------------------------------------------------------
```

Definition of Sequential LU Factorization

```
---------------------------------------------------------------------------
-- LU_Factorization Package Body (Sequential)
---------------------------------------------------------------------------

with Assertions; use Assertions;

package body LU_Factorization is

   ------------------------------------------------------------------------

   procedure LU_Factorize (A : in Matrix; LU : out Matrix) is
   begin
      Assert(A'Length > 0);
      Assert(A'First(1) = A'First(2) and A'Last(1) = A'Last(2));
      Assert(LU'First(1) = A'First(1) and LU'Last(1) = A'Last(1));
      Assert(LU'First(2) = A'First(2) and LU'Last(2) = A'Last(2));
      for I in LU'Range loop
         for J in LU'First .. I - 1 loop
            declare
               Sum : Float;
            begin
               Sum := 0.0;
               for K in LU'First .. J - 1 loop
                  Sum := Sum + LU(I, K)*LU(K, J);
               end loop;
               LU(I, J) := (A(I, J) - Sum)/LU(J, J);
            end;
         end loop;
         for J in I .. LU'Last loop
            declare
               Sum : Float;
            begin
               Sum := 0.0;
               for K in LU'First .. I - 1 loop
                  Sum := Sum + LU(I, K)*LU(K, J);
               end loop;
               LU(I, J) := A(I, J) - Sum;
            end;
         end loop;
      end loop;
   end LU_Factorize;

   ------------------------------------------------------------------------

end LU_Factorization;

---------------------------------------------------------------------------
```

B.4.2 Parallel LU Factorization using Barriers

Declaration of Barriers

```
-----------------------------------------------------------------------
-- Barriers Package Declaration
-----------------------------------------------------------------------

package Barriers is

   protected type Barrier (Num_Threads : Positive) is
      entry Reach_Barrier;
      entry Pass_Barrier;
   private
      All_Reached : Boolean := False;
      All_Passed  : Boolean := True;
      Count : Natural := 0;
   end Barrier;

   procedure At_Barrier (B : in out Barrier);

end Barriers;

-----------------------------------------------------------------------
```

Definition of Barriers

```
-----------------------------------------------------------------------
-- Barriers Package Body
-----------------------------------------------------------------------

package body Barriers is

   protected body Barrier is
      entry Reach_Barrier when All_Passed is
      begin
         Count := Count + 1;
         if Count = Num_Threads then
            All_Reached := True;
            All_Passed := False;
            Count := 0;
         end if;
      end Reach_Barrier;
      entry Pass_Barrier when All_Reached is
      begin
         Count := Count + 1;
         if Count = Num_Threads then
            All_Passed := True;
            All_Reached := False;
            Count := 0;
         end if;
      end Pass_Barrier;
   end Barrier;

   procedure At_Barrier (B : in out Barrier) is
```

```
   begin
      B.Reach_Barrier;
      B.Pass_Barrier;
   end At_Barrier;

end Barriers;

-----------------------------------------------------------------------
```

Declaration of Parallel LU Factorization using Barriers

```
-----------------------------------------------------------------------
-- LU_Factorization Package Declaration (Parallel/Barriers)
-----------------------------------------------------------------------

package LU_Factorization is

   type Matrix is array (Integer range <>,
                         Integer range <> ) of Float;

   procedure LU_Factorize (A           : in     Matrix;
                           LU          :    out Matrix;
                           Num_Threads : in     Positive);
   --| Requires:
   --|    A'Length > 0 and Square(A) and
   --|    Nonsingular(A) and Same_Bounds(LU, A) and
   --|    Num_Threads <= A'Length.
   --| Ensures:
   --|    Unit_Lower_Triangle(LU)*Upper_Triangle(LU) = A.

end LU_Factorization;

-----------------------------------------------------------------------
```

Definition of Parallel LU Factorization using Barriers

```
-----------------------------------------------------------------------
-- LU_Factorization Package Body (Parallel/Barriers)
-----------------------------------------------------------------------

with Assertions; use Assertions;
with Barriers; use Barriers;

package body LU_Factorization is

   --------------------------------------------------------------------

   procedure LU_Factorize (A           : in     Matrix;
                           LU          :    out Matrix;
                           Num_Threads : in     Positive) is

      function Split (First, Last, T : Integer) return Integer is
      begin
         return First + Integer(
            (Float(T)/Float(Num_Threads))*Float(Last - First + 1));
      end Split;

      B : Barrier(Num_Threads);

   begin
      Assert(A'Length > 0);
      Assert(A'First(1) = A'First(2) and A'Last(1) = A'Last(2));
      Assert(LU'First(1) = A'First(1) and LU'Last(1) = A'Last(1));
      Assert(LU'First(2) = A'First(2) and LU'Last(2) = A'Last(2));
      Assert(Num_Threads <= A'Length);
      parfor T in 0 .. Num_Threads - 1 loop
         for I in LU'Range loop
            for J in Split(I, LU'Last, T) ..
                     Split(I, LU'Last, T + 1) - 1 loop
               declare
                 Sum : Float;
               begin
                  Sum := 0.0;
                  for K in LU'First .. I - 1 loop
                     Sum := Sum + LU(I, K)*LU(K, J);
                  end loop;
                  LU(I, J) := A(I, J) - Sum;
               end;
            end loop;
            At_Barrier(B);
            for J in Split(I + 1, LU'Last, T) ..
                     Split(I + 1, LU'Last, T + 1) - 1 loop
               declare
                  Sum : Float;
               begin
                  Sum := 0.0;
                  for K in LU'First .. I - 1 loop
                     Sum := Sum + LU(J, K)*LU(K, I);
                  end loop;
                  LU(J, I) := (A(J, I) - Sum)/LU(I, I);
```

```
            end;
         end loop;
         At_Barrier(B);
      end loop;
   end loop;
end LU_Factorize;

-------------------------------------------------------------------

end LU_Factorization;

----------------------------------------------------------------------
```

B.4.3 Parallel LU Factorization using Single-Assignment Flags

Declaration of Single-Assignment Flags

```
-----------------------------------------------------------------------
-- Flags Package Declaration
-----------------------------------------------------------------------

package Flags is

   type Flag is limited private;

   procedure Set (F : out Flag);

   procedure Check (F : in Flag);

private

   type Flag is (Set);
   pragma Single_Assignment(Flag);

end Flags;

-----------------------------------------------------------------------
```

Definition of Single-Assignment Flags

```
-----------------------------------------------------------------------
-- Flags Package Body
-----------------------------------------------------------------------

package body Flags is

   procedure Set (F : out Flag) is
   begin
      F := Set;
   end Set;

   procedure Check (F : in Flag) is
   begin
      if F = Set then null; end if;
   end Check;

end Flags;

-----------------------------------------------------------------------
```

Declaration of Parallel LU Factorization using Single-Assignment Flags

```
------------------------------------------------------------------------
-- LU_Factorization Package Declaration (Parallel/Single-Assignment Flags)
------------------------------------------------------------------------

package LU_Factorization is

   type Matrix is array (Integer range <>,
                         Integer range <> ) of Float;

   procedure LU_Factorize (A           : in     Matrix;
                           LU          :    out Matrix;
                           Num_Blocks  : in     Positive;
                           Num_Threads : in     Positive );
   --| Requires:
   --|    A'Length > 0 and Square(A) and
   --|    Nonsingular(A) and Same_Bounds(LU, A) and
   --|    Num_Blocks <= A'Length and
   --|    Num_Threads <= Num_Blocks*Num_Blocks.
   --| Ensures:
   --|    Unit_Lower_Triangle(LU)*Upper_Triangle(LU) = A.

end LU_Factorization;

------------------------------------------------------------------------
```

Definition of Parallel LU Factorization using Single-Assignment Flags

```
-----------------------------------------------------------------------
-- LU_Factorization Package Body (Parallel/Single-Assignment Flags)
-----------------------------------------------------------------------

with Assertions; use Assertions;
with Flags; use Flags;

package body LU_Factorization is

   --------------------------------------------------------------------

   function Min (Left, Right : Integer) return Integer
   renames Integer'Min;

   function Max (Left, Right : Integer) return Integer
   renames Integer'Max;

   --------------------------------------------------------------------

   procedure LU_Factorize (A           : in     Matrix;
                           LU          :    out Matrix;
                           Num_Blocks  : in     Positive;
                           Num_Threads : in     Positive ) is

      function Start (I : Integer) return Integer is
      begin
         return LU'First +
            Integer((Float(I)/Float(Num_Blocks))*Float(LU'Length));
      end Start;

      B    : Integer;
      R, C : array (0 .. Num_Blocks*Num_Blocks - 1) of Integer;
      Done : array (-1 .. Num_Blocks - 1, -1 .. Num_Blocks - 1) of Flag;

   begin
      Assert(A'Length > 0);
      Assert(A'First(1) = A'First(2) and A'Last(1) = A'Last(2));
      Assert(LU'First(1) = A'First(1) and LU'Last(1) = A'Last(1));
      Assert(LU'First(2) = A'First(2) and LU'Last(2) = A'Last(2));
      Assert(Num_Blocks <= A'Length);
      Assert(Num_Threads <= Num_Blocks*Num_Blocks);
      B := 0;
      for I in 0 .. Num_Blocks - 1 loop
         for J in I .. Num_Blocks - 1 loop
            R(B) := I; C(B) := J; B := B + 1;
         end loop;
         for J in I + 1 .. Num_Blocks - 1 loop
            R(B) := J; C(B) := I; B := B + 1;
         end loop;
      end loop;
      for I in 0 .. Num_Blocks - 1 loop
         Set(Done(-1, I)); Set(Done(I, -1));
      end loop;
      pragma Parallelizable_Loop(Num_Threads, Pattern => On_Demand);
```

```
      for B in 0 .. Num_Blocks*Num_Blocks - 1 loop
         Check(Done(R(B), Min(R(B), C(B) - 1)));
         Check(Done(Min(C(B), R(B) - 1), C(B)));
         for I in Start(R(B)) .. Start(R(B) + 1) - 1 loop
            for J in Start(C(B)) .. Min(Start(C(B) + 1), I) - 1 loop
               declare
                  Sum : Float;
               begin
                  Sum := 0.0;
                  for K in LU'First .. J - 1 loop
                     Sum := Sum + LU(I, K)*LU(K, J);
                  end loop;
                  LU(I, J) := (A(I, J) - Sum)/LU(J, J);
               end;
            end loop;
            for J in Max(Start(C(B)), I) .. Start(C(B) + 1) - 1 loop
               declare
                  Sum : Float;
               begin
                  Sum := 0.0;
                  for K in LU'First .. I - 1 loop
                     Sum := Sum + LU(I, K)*LU(K, J);
                  end loop;
                  LU(I, J) := A(I, J) - Sum;
               end;
            end loop;
         end loop;
       Set(Done(R(B), C(B)));
      end loop;
   end LU_Factorize;

   ----------------------------------------------------------------------

end LU_Factorization;

-------------------------------------------------------------------------
```

Bibliography

[1] W. B. Ackerman and J. B. Dennis. VAL—a value-oriented algorithmic language: Preliminary reference manual. Technical Report TR-218, MIT Laboratory for Computer Science, Cambridge, Massachusetts, June 1979.

[2] William B. Ackerman. Data flow languages. *IEEE Computer*, 15(2):15–25, February 1982.

[3] *Ada 95 Reference Manual*. International Organization for Standardization, January 1995. International Standard ANSI/ISO/IEC-8652:1995.

[4] Eugene Albert, Joan D. Lukas, and Guy L. Steele, Jr. Data parallel computers and the FORALL statement. *Journal of Parallel and Distributed Computing*, 13(2):185–192, October 1991.

[5] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, Redwood City, California, second edition, 1994.

[6] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual*. Springer-Verlag, Berlin, Germany, 1983. ANSI/MIL-STD-1815A.

[7] Christiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[8] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, California, 1991.

[9] Arvind, K. P. Gostelow, and W. Plouffe. An asynchronous programming language and computing machine. Technical Report TR-114a, Department of Information and Computer Science, University of California, Irvine, December 1978.

[10] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. 1977 ACM Turing Award Lecture.

[11] Henri. E. Bal, Jennifer. G. Steiner, and Andrew. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[12] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[13] Forest Baskett and Alan Jay Smith. Interference in multiprocessor systems with interleaved memory. *Communications of the ACM*, 19(6):327–334, June 1976.

[14] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(6):85–97, March 1996.

[15] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[16] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, and Lawrence Rauchweger. Automatic detection of parallelism: A grand challenge for high-performance computing. *IEEE Parallel and Distributed Technology*, 2(3):37–47, Fall 1994.

[17] F. W. Burton. Functional programming for concurrent and distributed computing. *The Computer Journal*, 30(5):437–450, October 1987.

[18] Ralph M. Butler and Ewing L. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994.

[19] David Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[20] K. Mani Chandy and Ian Foster. A notation for deterministic cooperating processes. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):863–871, August 1995.

[21] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming language. Technical Report CS-TR-92-01, Computer Science Department, California Institute of Technology, 1992.

[22] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 281–313. MIT Press, Cambridge, Massachusetts, 1993.

[23] K. Mani Chandy and Stephen Taylor. A primer for Program Composition Notation. Technical Report CS-TR-90-10, Computer Science Department, California Institute of Technology, 1990.

[24] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, Boston, Massachusetts, 1992.

[25] Barbara Chapman, Hans Zima, and Piyush Mechrota. Extending HPF for advanced data-parallel applications. *IEEE Parallel and Distributed Technology*, 2(3):59–70, Fall 1994.

[26] A. Church and J. B. Rosser. Some properties of conversions. *Transactions of the American Mathematical Society*, 39:472–482, 1936.

[27] Keith Clark and Steve Gregory. PARLOG: Parallel programming in logic. Technical Report DOC 84/4, Department of Computing, Imperial College, London, April 1983.

[28] Keith Clark and Steve Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[29] Mark J. Clement and Michael J. Quinn. Overlapping computations, communications and I/O in parallel sorting. *Journal of Parallel and Distributed Computing*, 28(2):162–172, August 1995.

[30] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, Germany, third edition, 1984.

[31] Keith D. Cooper, Mary Hall, Ken Kennedy, and Linda Torczon. Interprocedural analysis and optimization. *Communications of Pure and Applied Mathematics*, 48(9–10):947–1003, September–October 1995.

[32] Keith D. Cooper, Mary W. Hall, Robert T. Hood, Ken Kennedy, Kathryn S. McKinley, John M. Mellorcrummey, Linda Torczon, and Scott K. Warren. The Parascope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[33] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, Inc., New York, New York, 1968.

[34] Jack Dongarra, David Walker, et al. Special issue – MPI – a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), Fall–Winter 1994.

[35] R. Kent Dybvig. *The SCHEME Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[36] D. J. Evans and N. Y. Yousif. Analysis of the performance of the parallel quicksort method. *BIT*, 25:106–112, 1985.

[37] J. T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, volume 6 of *Special Topics in Supercomputing*. North-Holland, Amsterdam, The Netherlands, 1992.

[38] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[39] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society*, pages 19–32. American Mathematical Society, 1967.

[40] High Performance Fortran Forum. High Performance Fortran language specification/journal of development. *Scientific Programming*, 2(1–2), Spring and Summer 1993.

[41] Ian Foster. Task parallelism and high-performance languages. *IEEE Parallel and Distributed Technology*, 2(3):27–36, Fall 1994.

[42] Ian Foster, Robert Olson, and Steven Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1):51–66, Fall 1992.

[43] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[44] Ian Foster and Stephen Taylor. A compiler approach to scalable concurrent-program design. *ACM Transactions of Programming Languages and Systems*, 16(3):577–604, May 1994.

[45] Ian T. Foster and K. Mani Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, April 1995.

[46] Christine Fricker. On memory contention problems in vector multiprocessors. *IEEE Transactions of Computers*, 44(1):92–105, January 1995.

[47] Mike Galles and Eric Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Proceedings of the 27th Annual IEEE Conference on Systems Science, Architecture Volume*, pages 134–144, Wailea, Hawaii, January 4–7 1994.

[48] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[49] E. W. Giering, Frank Mueller, and T. P. Baker. Features of the GNU Ada runtime library. In *Proceedings of ACM TRI-Ada '94*, pages 93–103, Baltimore, Maryland, November 6–11 1994.

[50] Ron Goldman and Richard P. Gabriel. Qlisp: Parallel processing in Lisp. *IEEE Software*, pages 51–59, July 1989.

[51] David Gries. *The Science of Programming.* Springer-Verlag, New York, New York, 1981.

[52] Thomas Gross, David R. O'Hallaron, and Jaspal Subhlok. Task parallelism in a high performance fortran framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, Fall 1994.

[53] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, December 3–8 1995.

[54] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[55] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. Scientific and Engineering Computation. MIT Press, Cambridge, Massachusetts, 1991.

[56] Philip J. Hatcher, Michael J. Quinn, Anthony J. Lapadula, Bradley K. Seevers, Ray J. Anderson, and Robert R. Jones. Data-parallel programming on MIMD computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

[57] W. Daniel Hillis. *The Connection Machine*. ACM Distinguished Dissertation. MIT Press, Cambridge, Massachusetts, 1985.

[58] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel programming. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[59] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[60] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[61] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[62] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the Association for Computing Machinery*, 21(2):277–292, April 1974.

[63] Paul Hudak. Para-functional programming. *IEEE Computer*, 19(8):60–71, August 1986.

[64] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[65] Paul Hudak. Para-functional programming in Haskell. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 159–196. ACM Press, New York, New York, 1991.

[66] Paul Hudak, Simon Peyton Jones, Philip Wadler, et al. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992.

[67] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

[68] T. Ito and R. H. Halstead, Jr., editors. *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1990. Proceedings of US/Japan Workshop on Parallel Lisp.

[69] Joseph JáJá and Pearl Y. Wang, editors. Special issue on data parallel algorithms and programming. *Journal of Parallel and Distributed Computing*, 21(1), April 1994.

[70] S. L. Peyton Jones. Parallel implementation of functional programming. *The Computer Journal*, 32(2):175–186, April 1989.

[71] J. L. W. Kessels. A conceptual framework for a nonprocedural programming language. *Communications of the ACM*, 20(12):906–913, December 1977.

[72] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, second edition, 1973.

[73] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.

[74] C. Koelbel, D. Loveman, R. Schrieber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, Massachusetts, 1994.

[75] Robert Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, July 1979.

[76] Leslie Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[77] A. H. Land and A. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28:495–520, 1960.

[78] Per S. Laursen. Simple approaches to parallel branch and bound. *Parallel Computing*, 19(2):143–152, February 1993.

[79] E. L. Lawler and D. E. Woods. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, July–August 1966.

[80] Jenq Kuen Lee and Dennis Gannon. Object oriented parallel programming experiments and results. In *Proceedings of Supercomputing '91*, pages 273–282, Albuquerque, New Mexico, November 18–22 1991.

[81] Daniel E. Lenoski and Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing.* Morgan Kaufmann, San Francisco, California, 1995.

[82] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[83] David J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. *ACM Computing Surveys*, 25(3):303–338, September 1993.

[84] John D. C. Little, Kalta G. Murty, Dura W. Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11(6):972–989, November–December 1963.

[85] W. Loots and T. H. C. Smith. A parallel algorithm for the 0-1 knapsack problem. *International Journal of Parallel Programming*, 21(5):349–362, October 1992.

[86] David May. Occam. *ACM SIGPLAN Notices*, 17(4):69–79, April 1983.

[87] J. R. McGraw, S. Allan, J. Glauert, and I. Dobes. SISAL: Streams and iteration in a single-assignment language, language reference manual. Technical Report M-146, Lawrence Livermore National Laboratory, 1983.

[88] James R. McGraw. The VAL language: Description and analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, January 1982.

[89] G. P. McKeown, V. J. Rayward-Smith, and S. A. Rush. Parallel branch-and-bound. In Lydia Kronsjö and Dean Shumsheruddin, editors, *Advances in Parallel Algorithms*, chapter 5, pages 111–150. Halsted Press, John Wiley and Sons, New York, 1992.

[90] Michael Metcalf and John Reid. *Fortran 90 Explained.* Oxford University Press, Oxford, Great Britain, 1990.

[91] Robin Milner, Mads Tofte, and Robin Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[92] L. G. Mitten. Branch-and-bound methods: General formulation and properties. *Operations Research*, 18(1):24–34, January–February 1970.

[93] Greg Nelson, editor. *Systems Programming with Modula-3.* Innovative Technology. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[94] Ridhiyur S. Nikhil and Arvind. Id: a language with implicit parallelism. In J. T. Feo, editor, *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, volume 6 of *Special Topics in Supercomputing*, pages 169–215. North-Holland, Amsterdam, The Netherlands, 1992.

[95] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[96] Cherri M. Pancake. Multithreaded languages for scientific and technical computing. *Proceedings of the IEEE*, 81(2):288–304, February 1993.

[97] *Draft Standard for Information Technology – Portable Operating Systems Interface (POSIX).* IEEE, September 1994. P1003.4a/D10.

[98] William H. Press, Saul A Teukolsky, William T. Vetterling, and Brian P. Flannery, editors. *Numerical Recipes in C.* Cambridge University Press, Cambridge, Great Britain, second edition, 1992.

[99] Michael J. Quinn. Analysis and implementation of branch-and-bound algorithms on a hypercube multicomputer. *IEEE Transactions on Computers*, 39(3):384–387, March 1990.

[100] Michael J. Quinn and Philip J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, pages 69–76, September 1990.

[101] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993.

[102] John R. Rose and Guy L. Steele, Jr. C*: An extended C language for data parallel programming. In *Proceedings of the Second International Conference on Supercomputing*, volume 2, pages 2–16, May 1987.

[103] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, December 1991.

[104] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, California, November 14–17 1994.

[105] Edmond Schonberg and Bernard Banner. The GNAT project: A GNU-Ada 9X compiler. In *Proceedings of ACM TRI-Ada '94*, pages 48–57, Baltimore, Maryland, November 6–11 1994.

[106] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, October 1978.

[107] Robert Sedgewick. *Algorithms.* Addison-Wesley, Reading, Massachusetts, second edition, 1988.

[108] Symmetric multiprocessing. Technical report, Silicon Graphics, Inc., 1994.

[109] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, ICOT, Institute for New Generation Computer Technology, Tokyo, Japan, 1983.

[110] Ehud Shapiro. Concurrent Prolog: A progress report. *IEEE Computer*, 19(8):44–58, August 1986.

[111] Ehud Shapiro, editor. *Concurrent Prolog: Collected Papers*, volume 1. MIT Press, Cambridge, Massachusetts, 1987.

[112] Ehud Shapiro, editor. *Concurrent Prolog: Collected Papers*, volume 2. MIT Press, Cambridge, Massachusetts, 1987.

[113] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

[114] Hanmao Shi and Jonathon Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, April 1992.

[115] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference.* MIT Press, Cambridge, Massachusetts, 1995.

[116] Richard Stallman. *Using and Porting GNU GCC.* Free Software Foundation, Cambridge, Massachusetts, 1994.

[117] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[118] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23(5):54–64, May 1990.

[119] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, April 1994.

[120] Boleslaw K. Szymanski, editor. *Parallel Functional Languages and Compilers.* ACM Press, New York, New York, 1991.

[121] Stephen Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, Engelwood Cliffs, New Jersey, 1989.

[122] Stephen Taylor, Jerrell Watts, Marc Rieffel, and Michael Palmer. The concurrent graph: Basic technology for irregular problems. *IEEE Parallel and Distributed Technology*, 1996.

[123] L. G. Tesler and H. J. Enea. A language design for concurrent processes. In *Proceedings of the 1968 AFIPS Spring Joint Computer Conference*, pages 403–408, Atlantic City, New Jersey, April 30–May 2 1968.

[124] John Thornley. Parallel programming with Declarative Ada. Technical Report CS-TR-93-03, Computer Science Department, California Institute of Technology, 1993.

[125] John Thornley. Integrating functional and imperative parallel programming: CC++ solutions to the Salishan problems. In *Proceedings of the 8th IEEE International Parallel Processing Symposium (IPPS '94)*, pages 61–67, Cancún, Mexico, April 26–29 1994.

[126] John Thornley. Integrating parallel dataflow programming with the Ada tasking model. In *Proceedings of ACM TRI-Ada '94*, pages 417–428, Baltimore, Maryland, November 6–11 1994.

[127] John Thornley. Declarative Ada: Parallel dataflow programming in a familiar context. In *Proceedings of the 23rd Annual ACM Computer Science Conference (CSC '95)*, pages 73–80, Nashville, Tennessee, February 28–March 2 1995.

[128] John Thornley. Performance of a class of highly-parallel divide-and-conquer algorithms. Technical Report CS-TR-95-10, Computer Science Department, California Institute of Technology, 1995.

[129] John Thornley. Performance of a high-level parallel programming layer defined on top of the Ada tasking model. In *Proceedings of TRI-Ada '95*, pages 252–262, Anaheim, California, November 5–10 1995.

[130] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[131] D. A. Turner. The semantic elegance of applicative languages. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 85–92, Portsmouth, New Hampshire, October 1981.

[132] Eric F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag, New York, New York, 1994.

[133] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoft, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5(1–3):1–236, 1975.

[134] Paul G. Whiting and Robert S. V. Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16(4):38–59, Winter 1994.

[135] Niklaus Wirth. A note on "Program Structures for Parallel Processing". *Communications of the ACM*, 9(5):320–321, May 1966.

[136] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, California, 1996.

[137] Myung K. Yang and Chita R. Das. Evaluation of a parallel branch-and-bound algorithm on a class of multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):74–86, January 1994.