RESEARCH MONOGRAPHS IN PARALLEL AND DISTRIBUTED COMPUTING

Edited by
David Gelernter, Yale University,
Alexandru Nicolau, University of California, Irvine, and
David Padua, University of Illinois at Urbana-Champaign

Languages and Compilers for Parallel Computing

RESEARCH MONOGRAPHS IN PARALLEL AND DISTRIBUTED COMPUTING

Edited by
David Gelernter, Yale University,
Alexandru Nicolau, University of California, Irvine, and
David Padua, University of Illinois at Urbana-Champaign

Languages and Compilers for Parallel Computing

Pitman, London

The MIT Press, Cambridge, Massachusetts

PITMAN PUBLISHING
128 Long Acre, London WC2E 9AN

First published 1990

Available in the Western Hemisphere and Israel from
The MIT Press
Cambridge, Massachusetts (and London, England)

ISSN 0953-7767

British Library Cataloguing in Publication Data
Languages and compilers for parallel computing.—(Research monographs in parallel and distributed computing.
ISSN 0953-7767)
1. Computer systems. Parallel-processor systems. *Programming languages*
I. Gelernter, David II. Nicolau, Alexandru III. Padua, David IV. Series
005.4'5

ISBN 0-273-08820-3

Library of Congress Cataloging-in-Publication Data
Languages and compilers for parallel computing / edited by David Gelernter, Alexandru Nicolau, and David Padua.
p. cm.—(Research monographs in parallel and distributed computing. ISSN 0953-7767)
Includes bibliographical references.
ISBN 0-262-57080-7
1. Programming languages (Electronic computers) 2. Compilers (*Computer programs*) 3. *Parallel processing* (Electronic computers)
I. Gelernter, David Hillel. II. Nicolau, Alexandru. III. Padua, David A. IV. Series.
QA76.7.L38 1990
0054'53—dc20

Reproduced and printed by photolithography
in Great Britain by Biddles Ltd, Guildford

Foreword

This book contains a selection of the papers presented at the Second Workshop on Languages and Compilers for Parallel Computing which took place in Urbana, Illinois during the first three days of August 1989. This workshop was sponsored by the Center for Supercomputing Research and Development of the University of Illinois at Urbana-Champaign. The First Workshop of this series took place at Cornell University in 1988, and there are plans to hold a third Workshop in 1990.

The topic of the papers in this book is today of great interest in the research and industrial communities. A manifestation of this interest was the large number of participants, representing many different institutions, in the Workshop. This interest, which only a decade ago was confined to a few research groups, has been spurred by the widespread proliferation of vector and parallel computers that has taken place during the last several years. The need for research in these areas is commonly accepted since, despite all the progress made, the question of what programming languages should be used has not been settled, and much remains to be developed in the area of compiler techniques.

These Proceedings contain several papers discussing languages and language extensions for parallel computing, including the papers by Baldwin; Callahan and Smith; Chandra et al.; Ciancarini; Klappholz et al.; Snyder; and Solworth. There are also a few papers that describe interactive/graphical environments that extend or complement traditional programming languages (Bailey and Cuny; Browne). In the area of compilers and restructurers, the papers contained in this book cover several topics, including: fundamental parallelization techniques and parallelization systems (Banerjee; Cytron et al.; Li and Yew; Polychronopoulos et al.), techniques for fine-grain parallelism (Aiken and Nicolau; Ebcioglu and Nakatani; Uht), for shared-memory parallel processors (Rochat), for distributed memory parallel processors (Wolfe), and for dataflow computers (Gao and Paraskevas). There are also two papers on tools for parallel programming, one for debugging (Choi and Miller), and the other for performance enhancement (Kwan et al.).

Most of the existing parallelizing compilers are aimed at Fortran, but parallelization techniques are being rapidly developed for other languages. In this book, work is reported on the parallelization of C (Gannon et al.) and Lisp (Harrison and Ammarguellat). Finally, in the area of compilation and restructuring of parallel programs there are papers on the translation of C-Linda (Carriero and Gelernter), machine code optimization for the Cray computer (Eisenbeis et al.), and techniques for the further parallelization of parallel programs (Midkiff et al.).

We believe this book presents a good panoramic view of the state of the research in Languages and Compilers for Parallel Computing in 1989. We hope that this book will be as exciting for you to read as it was for us to compile.

David Padua

Contents

1 Fine-grain Parallelization and the Wavefront Method

Alexander Aiken and Alexandru Nicolau

Abstract

We develop a technique for extracting parallelism from ordinary (sequential) programs. The technique combines two fine-grain, instruction level code transformations to achieve effects similar to the wavefront method. Such effects were previously available only as transformations at coarser levels of granularity. By integrating the strengths of the wavefront method with the ability to extract fine-grain, irregular parallelism at the instruction level, our technique exploits previously untapped parallelism.

1 Introduction

Progress in parallelizing compiler technology is perhaps the most important factor in translating the promise of parallel computing—dramatically faster computation—into reality for most users. Much progress has been made, and many techniques developed for extracting parallelism from ordinary programs [5,7,10,11,18] have been integrated into production tools.

Compile-time parallelization techniques fall roughly into two, largely disjoint classes. The first class, fine-grain, low-level parallelization, extracts irregular parallelism at the instruction level, but cannot deal well with parallelism at the level of loops[1]. The second class sacrifices irregular parallelism in favor of high-level, regular parallelism

[1] While simple loop-unwinding may alleviate this problem, it does not eliminate it.

achieved by overlapping (partially or completely) loop iterations or full loops. Percolation Scheduling [17] is an example of the first class of techniques; doacross [8] is an example of the second class. The strengths of these two approaches are complimentary; with the emergence of machines that can exploit both instruction-level and coarser forms of parallelism—such as Multiflow's Trace, Cydrome's Cydra, Chopp, Alliant, Burton Smith's Horizon, and machines based on Intel's i860 chip—the integration of these levels of parallelism becomes important.

We show how to uniformly integrate fine-grain and coarse-grain parallelization of nested loops. We make use of two techniques: perfect pipelining [1,4], a transformation that bridges the gap between instruction level and iteration level parallelization for innermost loops, and loop quantization [3,16], a transformation that exposes instruction-level parallelism across nested loops. The algorithm shares many of the properties of the wavefront method [19], one of the most powerful high-level (nested-loop) parallelizing transformations, while also exploiting any irregular parallelism available at the fine-grain (instruction) level. The development illustrates the surprising expressive power of the fine-grain transformations and their relation to the wavefront method.

2 Basic Definitions

A set of nested loops L consists of loops labeled L_1 (outermost) to L_n (innermost). Each loop L_j has an associated index variable I_j. The *iteration space* of L consists of *iteration vectors* $\langle i_1, \ldots, i_n \rangle$ where each i_j is an assignment to I_j [13]. Iteration vectors are ordered lexicographically; this corresponds to the normal sequential execution order of iterations. We assume that $\langle 0, \ldots, 0 \rangle$ is the first iteration of any loop.

Programs are represented as *program-graphs* (also called control-flow graphs) where nodes contain zero or more statements. A statement is either an *assignment* or a *test* (an If-statement). Execution of a program begins at a distinguished start node and proceeds sequentially from node to node. A node is executed by evaluating the node's statements in parallel; the assignments update the store and the tests return the next node to be executed. The precise definition of the concurrent execution of tests is beyond the scope of this paper; details may be found in [1,17].

Program parallelization requires *dependency analysis*. Two program statements are dependent if one accesses a storage location that the other writes. Dependent statements may not be executed in parallel. Dependency information is usually represented

by a *dependency graph*, where an edge between two statements represents a potential dependency [12]. We say that two iterations of a loop are dependent if any pair of statements from the two iterations is dependent.

Several of the examples include a picture of the iteration space and the dependencies between iterations. The iteration space is depicted in the first quadrant of the plane; points in the plane are iterations. The horizontal axis represents the inner loop, the vertical axis represents the outer loop. An arrow is drawn between two iterations if they are dependent; thus, an arrow simply means that the normal order of execution between the iterations must be enforced. For simplicity, we use only one type of dependency. In the literature, dependencies are usually classified as belonging to one of several types [12], but this is unnecessary for the presentation of our technique.

Our algorithm uses three low-level transformations of a program-graph: *unroll*, *move*, and *delete*. *Unroll* adds (unrolls) one copy of the original loop body at the end of the current loop body. We distinguish between the original and current loops because we alternately unroll and perform code motions. The *move* transformation moves a statement x from a node i to a predecessor of i if dependencies are not violated. The *delete* transformation deletes an empty node (a node with no statements) from the program-graph. A program is parallelized by application of these transformations, packing multiple statements into nodes for parallel execution. These transformations are based on the primitives of percolation scheduling. Complete descriptions of the transformations and the model of computation are in [2].

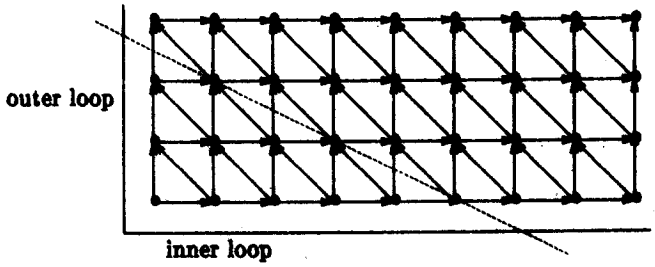
For simplicity, we restrict the development to two nested loops with a single assignment in the loop body. The results apply directly to loops with any number of statements and arbitrary flow-of-control. Because of the simple flow-of-control in the example loops, we can adopt a representation more readable than program graphs. Statements are written in a standard high-level syntax. All statements on each line of a program are executed in parallel; successive lines are executed sequentially.

3 The Wavefront Method

The wavefront method [13,14,15,19,20] extracts parallelism from multiple nested loops in many cases where parallelism cannot be found in any single loop. The traditional derivation of the wavefront method involves finding a legal wavefront (i.e., a line in two dimensions, a plane or hyperplane in higher dimensions) through the iteration space.

```
for i ← 0 to N_j do
    for j ← 0 to N_i do
        A[i,j] ← A[i − 1, j + 1] + A[i − 1, j] + A[i, j + 1];
```

(a) A sample loop.



(b) The iteration space and optimal wavefront.

Figure 1: An example of the wavefront method.

All iterations on the same wavefront may be executed in parallel. A legal wavefront preserves dependencies—two dependent iterations are executed in the same order as in the original loop. When a loop is executed in the wavefront ordering, iteration $\langle i, j \rangle$ is executed at wavefront (time) step $i * b + j$, where the wavefront angle is $\arctan(-1/b)$.

The goal of the wavefront method is to find a wavefront that maximises parallelism; that is, the goal is to find a wavefront that maximises the number of iterations executed in parallel at each step. Consider the loop in Figure 1a. Figure 1b shows the iteration space and dependencies; the optimal wavefront angle is 30°. We use a slightly different formulation of the wavefront to make comparison with our technique direct. The wavefront is expressed as a line with slope $-1/b$ for some positive integer b. (As in [19], we do not consider the cases where the wavefront angle is 0° or 90°. In these cases, the wavefront method does not apply.) There are many potential wavefronts that cannot be expressed by these ratios; however, the optimal wavefront is always of this form. In Figure 1b, the optimal wavefront slope is $-1/2$.

The wavefront must preserve dependencies; a wavefront that is too steep may reverse the order of dependent iterations. The following definition describes all legal wavefronts.

Definition 3.1 (Legal Wavefronts) Let L be two nested loops, let $-1/b$ be a wavefront slope, and let $\langle i, j \rangle$ and $\langle i', j' \rangle$ be the iteration vectors of any dependent iterations.

Then the wavefront is legal if and only if:

$$\langle i,j\rangle < \langle i',j'\rangle \Rightarrow i*b+j < i'*b+j'$$

4 Loop Quantization

Loop Quantization [16] is a technique for unrolling multiple nested loops. Loop unrolling provides a large number of instructions—the unrolled loop body—for scheduling by instruction-level transformations (such as trace scheduling [9] or percolation scheduling [2,17]). Unrolling nested loops is important because parallelism may be present in outer loops and not in the inner loop, or even across several of the nested loops.

In general, loop quantization computes integers $k_1,\ldots,k_n$, where n is the number of nested loops. The original loop is unrolled k_1 times on the innermost loop, then this new loop body is unrolled k_2 times on the next innermost loop, and so on. The resulting loop $\hat{L}$ has an n-dimensional "box" of iterations of L as its loop body. All iterations in the box are executed before the box is shifted (by a "quantum" jump) along any of the dimensions. An example of a two by two quantization of the loop in Figure 1 is given in Figures 2a and 2b.[2]

Quantization alters the execution order of iterations of L and may therefore violate data dependencies. Conditions under which a quantization is legal are given in [1,3]. The two by two quantization given in Figure 2b is illegal. As shown in Figure 3, the quantization box "cuts" a dependency illegally—iteration $\langle i,j\rangle$ must execute before iteration $\langle i+1,j-1\rangle$.

Under certain conditions a *mitred quantization* can permit quantization where a normal quantization is illegal [1,3]. Mitred quantization permits rhomboid quantization boxes instead of simple rectangular quantization boxes. For example, in Figure 4, the two by two mitred quantization is legal, because all dependent iterations are either included inside a quantized iteration or are satisfied by normal loop order execution in the quantized iteration space. Figure 2c shows the example loop after a two by two mitred quantization.

Mitred quantization computes the slant of the quantization box from the dependencies of the loops. For instance, in Figure 1, iteration $\langle i,j\rangle$ is dependent on iteration $\langle i+1,j-1\rangle$. The *slope* of this dependency is $-1/1$. Mitred quantization selects as the

[2]Quantised loops require a small amount of extra code to handle boundary conditions; for details see [1,3].

```
for i ← 0 to N_i do
    for j ← 0 to N_j by 2 do
        begin
            A[i,j] ← A[i − 1,j + 1] + A[i − 1,j] + A[i,j + 1];
            A[i,j + 1] ← A[i − 1,j + 2] + A[i − 1,j + 1] + A[i,j + 2];
        end
```

(a) Loops after inner loop is unrolled.

```
for i ← 0 to N_i by 2 do
    for j ← 0 to N_j by 2 do
        begin
            A[i,j] ← A[i − 1,j + 1] + A[i − 1,j] + A[i,j + 1];
            A[i,j + 1] ← A[i − 1,j + 2] + A[i − 1,j + 1] + A[i,j + 2];
            A[i + 1,j] ← A[i,j + 1] + A[i,j] + A[i + 1,j + 1];
            A[i + 1,j + 1] ← A[i,j + 2] + A[i,j + 1] + A[i + 1,j + 2];
        end
```

(b) Loops after 2×2 quantization.

```
for i ← 0 to N_i by 2 do
    for j ← 0 to N_j by 2 do
        begin
            A[i,j + 1] ← A[i − 1,j + 2] + A[i − 1,j + 1] + A[i,j + 2];
            A[i,j + 2] ← A[i − 1,j + 3] + A[i − 1,j + 2] + A[i,j + 3];
            A[i + 1,j] ← A[i,j + 1] + A[i,j] + A[i + 1,j + 1];
            A[i + 1,j + 1] ← A[i,j + 2] + A[i,j + 1] + A[i + 1,j + 2];
        end
```

(c) Loops after 2×2 mitred quantization.

Figure 2: How quantization works.

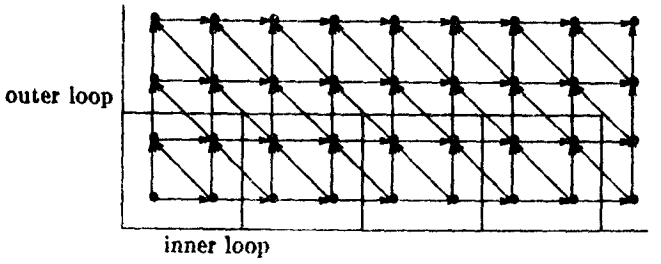


Figure 3: The two by two quantization is illegal.

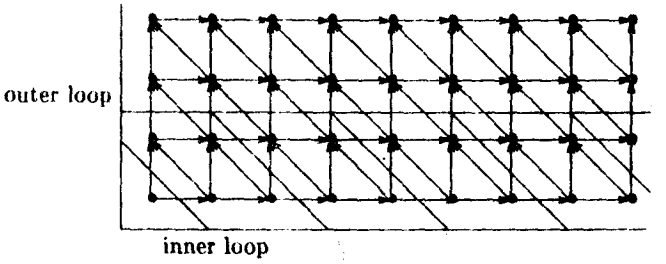


Figure 4: A legal mitred quantization of the loop in Figure 1.

slope of the sides of the quantization box the smallest negative slope of all dependencies. This produces a non-trivial, legal quantization for loops encountered in practice. As the following Lemma shows, there is a strong relationship between mitred quantization and the wavefront method. Indeed, similar effects to mitred quantizations can be achieved by combining the unroll-and-jam of [6] with the skewing produced by [19].

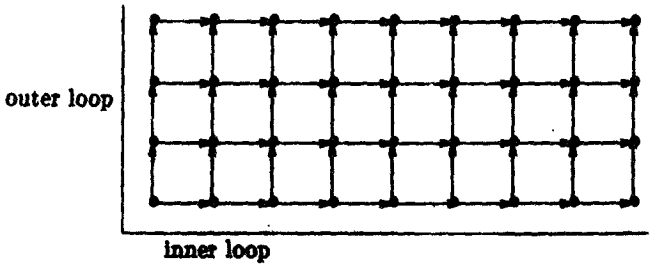
Lemma 4.1 Let L be two nested loops. Then the following statements about L are equivalent:

1. The optimal wavefront slope is $-1/b$.
2. The smallest negative slope of any dependency is $-1/(b-1)$.
3. A mitred quantization box has slope $-1/(b-1)$.

Proof: Let $-1/(b-1)$ be the slope of the mitred quantization. By definition, this is also the smallest negative slope of any dependency. It is easy to show that the wavefront slope $1/b$ is legal by Constraint 3.1, because it preserves the dependency with the smallest negative slope. Furthermore, this is the optimal wavefront slope--a slope of $1/(b-1)$ would put two dependent iterations on the same wavefront. Finally, if $1/b$ is

```
for i ← 0 to N_i
    for j ← 0 to N_j
        A[i,j] ← (A[i + 1,j] + A[i,j + 1] + A[i − 1,j] + A[i,j − 1])/4;
```

(a) Gauss-Seidel iteration.



(b) The iteration space.

Figure 5: Another loop.

the optimal wavefront slope, then there must be a dependency of slope $-1/(b-1)$ and no dependency of smaller negative slope. If the optimal wavefront slope is $-1/1$, then normal (rectangular) quantisation applies. □

5 Perfect Pipelining

Quantisation alone is insufficient to express the wavefront method using fine-grain parallelisation. In principle, there is a fundamental difference. The wavefront method expresses unbounded parallelism—the number of iterations executable in parallel in the wavefront method is unrestricted, and parallelism grows with the size of the iteration space. In our code compaction model, each node in the program graph can contain only a finite number of statements. However, this is not a meaningful difference; programs must be executed on a machine with fixed resources—e.g., processors. We make the assumption that there is a k such that no more than k iterations can be executed in parallel. We derive an algorithm that exposes at least as much parallelism as the wavefront method for any machine size k.

There is a more serious barrier to achieving the effect of the wavefront method using fine-grain parallelisation. Consider the standard Gauss-Seidel iteration loop in Figure 5; the iteration space with dependencies is shown in Figure 6. Assume that

```
for i ← 0 to N_i by 3 do
    for j ← 0 to N_j by 4 do
        begin
            A[i,j] ← (A[i + 1,j] + A[i,j + 1]+
                A[i − 1,j] + A[i,j − 1])/4;
            A[i,j + 1] ← (A[i + 1,j + 1] + A[i,j + 2]+
                A[i − 1,j + 1] + A[i,j])/4;
            ⋮
            A[i,j + 3] ← (A[i + 1,j + 3] + A[i,j + 4]+
                A[i − 1,j + 3] + A[i,j + 2])/4;
            ⋮
            A[i + 2,j + 2] ← (A[i + 3,j + 2] + A[i + 2,j + 3]+
                A[i + 1,j + 2] + A[i + 2,j + 1])/4;
            A[i + 2,j + 3] ← (A[i + 3,j + 3] + A[i + 2,j + 4]+
                A[i + 1,j + 3] + A[i + 2,j + 2])/4
        end;
```

Figure 6: Loop with three by four quantization.

```
for i ← 0 to N_i by 3 do
    for j ← 0 to N_j by 4 do
        begin
            [i,j]
            [i,j + 1]       [i + 1,j]
            [i,j + 2]       [i + 1,j + 1]   [i + 2,j]
            [i,j + 3]       [i + 1,j + 2]   [i + 2,j + 1]
            [i + 1,j + 3]   [i + 2,j + 2]
            [i + 2,j + 3]
        end;
```

Figure 7: Loop after quantization and compaction.

```
for i ← 0 to N_i by 3 do
    for j ← 0 to N_j by 6 do
        begin
            [i,j]
            [i,j+1]     [i+1,j]
            [i,j+2]     [i+1,j+1]   [i+2,j]
            [i,j+3]     [i+1,j+2]   [i+2,j+1]
            [i,j+4]     [i+1,j+3]   [i+2,j+2]
            [i,j+5]     [i+1,j+4]   [i+2,j+3]
            [i+1,j+5]   [i+2,j+4]
            [i+2,j+5]
        end;
```

Figure 8: Loop after further unrolling and compaction.

the target machine has sufficient resources to execute three iterations of this loop in parallel. The loop unrolled with a three by four quantization is given in Figure 7.[3] (We show below how to determine the amount of such unrollings.) The loop after compaction—every statement moved as far "up" as possible using move and delete—is given in Figure 8. For brevity, only the values of the index variables are given for each statement. Although full resource utilization is achieved in the middle of the loop body, utilization at the beginning and the end of the loop body is lower. Furthermore, regardless of the unrolling on either loop, after compaction there is always some start-up and wind-down code in the loop body.

The wavefront method also has start-up and wind-down periods where resources are less than fully utilized. However, for the wavefront method this occurs only once at the beginning and once at the end of the execution of the nested loops. Simply quantising and compacting produces code that under-utilises resources at the beginning and end of every quantized iteration.

If a loop could be fully unrolled and compacted (making the quantization box the entire iteration space of that loop) then the start-up and wind-down costs would be incurred only at the beginning and end of the loop's execution. Full unrolling is generally undesirable or impossible; however, there is a transformation, *perfect pipelining*, which achieves the effect of full unrolling and compaction of a loop[1,4]. Perfect Pipelining combines very fine-grain parallelism with the pipelining of loop iterations. The idea behind perfect pipelining is that a loop's dependencies encode some repeating behav-

[3]Normal quantisation is adequate for this example.