# Achieving Robust Performance in Parallel Programming Languages

E Christopher Lewis

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2001

Program Authorized to Offer Degree: Department of Computer Science and Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

E Christopher Lewis

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

_____

Lawrence Snyder

Reading Committee:

_____

Craig Chambers

_____

Burton Smith

_____

Lawrence Snyder

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the Doctorial degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with “fair use” as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted “the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform.”

Signature_____

Date_____

University of Washington

Abstract

# Achieving Robust Performance in Parallel Programming Languages

E Christopher Lewis

Chair of Supervisory Committee:

Professor Lawrence Snyder
Department of Computer Science and Engineering

Despite more than two decades of research effort, the question remains: *how can we realize the potential of large-scale parallel machines?* It *can* be done now, but only at great expense (*i.e.*, development time and effort) and with limited portability, rendering the exploitation of parallelism impractical for most users. Advanced–ZPL (A–ZPL) is a parallel programming language intended to address this problem. It's design was guided by a predictive performance model that clearly defines the role of the programmer and the compiler, called the programmer-compiler separation. The former is responsible for abstract parallel and sequential algorithmic issues, while the latter manages the tractable elements of mapping abstract representations to a particular machine. This dissertation evaluates the design and implementation of A–ZPL in the light of this design criteria. Specifically, we examine two aspects of the language and the compiler implications: efficient loop generation and pipelining wavefront computations. We find the language is highly effective both relatively and absolutely as a direct consequence of considering the programmer-compiler separation.

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## ACKNOWLEDGMENTS

I thank my friends and colleagues who make life such a joy. I will enumerate some of the more superlative examples. Brad Chamberlain, Geoff Voelker, Alec Wolman, Mike Perkowitz, Dave Grove, Matthai Philipose, Ted Romer, Doug Zongker, Brett Allen, Luke McDowell, Omid Madani, Jason Secoski, and Ton Ngo never shrink from an opportunity to talk shop. Ari Ellis, Rob Hadley, and Jim Park always remind me how great people can be. Sung-Eun Choi, Jim Fix, and Ben Dugan have been very tolerant of the idiosyncrasies of the imp that I was and continue to be. Calvin Lin has taught me the virtues of hard work, and Larry Snyder has helped me discover discovery. They are all friends.

## DEDICATION

I dedicate this work to my mother Cordelia, my father Edward, and my sister Heidi.

# Chapter 1

# INTRODUCTION

Despite more than two decades of research effort, the question remains: *how can we realize the potential of large-scale parallel machines?* Countless parallel platforms have been constructed, and particular applications have demonstrated each machine's peak performance but only after great development effort. A general and truly practical system for parallel development has yet to emerge. Although it has been long-held that parallelism is the preeminent method for improving the performance of future systems, significant machine and human resources are currently wasted in pursuit of parallelism. New and effective parallel programming techniques are essential for exploiting parallelism. The following excerpt from recent Compaq AlphaServer SC promotional literature characterizes the current morass of approaches to parallel programming [Hig00].

> Typically, programs will be written using Compaq FORTRAN, Compaq C, or Compaq C++. Compaq Fortran and Compaq C both include support for the industry-standard OpenMP directives, which are widely used for parallel development on shared-memory SMP systems.
>
> Most large-scale jobs that use the full capability of the *AlphaServer* SC system will be programmed using a message-passing model. . . via the Message Passing Interface (MPI) library. In some cases, higher performance will be achieved through the deployment of a hybrid-programming model, where a set of shared address space threaded processes (written, for example, with OpenMP) pass messages *via* MPI. For the highest possible performance, but at the expense of additional complexity, programmers can use the shared-memory Shmem facility to take the fullest advantage of the *AlphaServer* SC Interconnect's extremely low-latency one-sided communication capability. Yet another alternative. . ., the Compaq FORTRAN compiler includes High Performance Fortran (HPF).

As implied by the excerpt, users must weigh poorly defined issues—such as desired performance, the programming effort they are willing to expend, and scale of application—before selecting a system for developing a parallel application. This state of affairs renders parallel programming impractical for most users. Our goal is a system for *practical* parallel programming. The remainder

of the introduction examines the impediments to practical parallel programming, presents a general approach to addressing the impediments, and introduces an instantiation of this approach in the Advanced-ZPL parallel programming language.

### *1.1 Impediments to Practical Parallel Programming*

Good execution performance is the virtually unanimous goal of those who use parallelism. But most potential users lack the resources (time, programmers, knowledge of parallel programming and machines, *etc*.) required to pursue parallel performance when it comes at the expense of portability and usability. Thus, for a parallel programming system to be practical, it must provide *performance*, *portability*, and *usability*.

A programming system is considered portable when the mutually relative performance of any two codes is roughly independent of the target architecture. For example, in the sequential domain, the C programming language is considered portable; for the manifestly superior of two programs will be better on most—if not all—serial machines. Conversely, languages such as Prolog are less portable, for performance may vary widely—from fast to intractable—between different compilers or run-time systems. That an application merely runs *correctly* on different machines is a trivial form of portability; thus, we use the more meaningful definition that includes performance. A sufficient number and diversity of parallel platforms exist and are being developed that portability represents a significant practical concern. Just as most users and developers of serial machines are unwilling to invest in software that dies with the machine on which it is originally implemented, so, too, are users of parallel machines.

Usability is another important property of a programming system. One system is considered more usable than another when the former requires the dedication of less programming effort to particular implementation details so that the programmers are freed to reason about their codes at a higher level. Thus, C has a higher degree of usability than assembly language, for the latter requires the programmer to manage details such as register allocation, explicit call-stack management, and so on. A parallel programming system that lacks usability is unlikely to achieve general user acceptance. Although some will demand parallelism at any cost (even with poor usability), they will be members of a community not large enough to support code reuse and the development of advanced

tools and environments. Performance at the expense of usability and portability greatly limits the value of parallelism.

It would appear that performance, usability, and portability are all at odds with each other. The *PUP Triangle* in Figure 1.1 graphically depicts their conflicting relationships. In the diagram, the three properties—portability, usability, and performance—are arranged in a triangle, and arrows are labeled with a description of the principal means for achieving the properties to which they point. Opposing arrows denote conflicting means.

Consider the issues of performance and portability. Good performance is often achieved by *specializing* a program to particular architectural features. For example, the Compaq AlphaServer SC—the subject of the quotation, above—and the Cray T3E provide low-latency, one-sided communication; thus, one would like to use the Cray Shared Memory Access Library (Shmem) to exploit it [BK94]. This is not a simple matter of selecting a particular library; rather, it is intimately tied to the code structure ultimately developed. At the same time, portability is often achieved by *generalizing* a code's implementation and operation. Specialization and generalization are clearly at odds, implying the same for performance and portability. In the example, a well designed program written using Shmem may perform well on the Cray T3E, but it will almost certainly have abysmal performance on a machine with different features, such as the IBM SP, which has a higher latency and lower bandwidth network than the T3E or the AlphaServer.

Next, consider performance and usability. Specialization for performance often requires an extensive understanding of esoteric, machine-specific features. But usability is best achieved via *simplicity* of representation. The details of specialization put performance at odds with usability.

Finally, consider usability and portability. Usability is best achieved by limiting the complexity of the programmer's view of a computation, while portability is best achieved via generalization. Generalization for portability—with performance—serves to make the performance of an application insensitive to particular machine features. Such generalization requires a deep understanding of the abstract properties of parallel machines. This knowledge and the effort to exploit it represent details of implementation rather than algorithmic fundamentals. For this reason, simplification and generalization—and usability and portability—are at odds.

To be concrete, the following exemplify the tension between portability and both performance and usability. First, the Message Passing Interface (MPI) library is billed as a portable system for

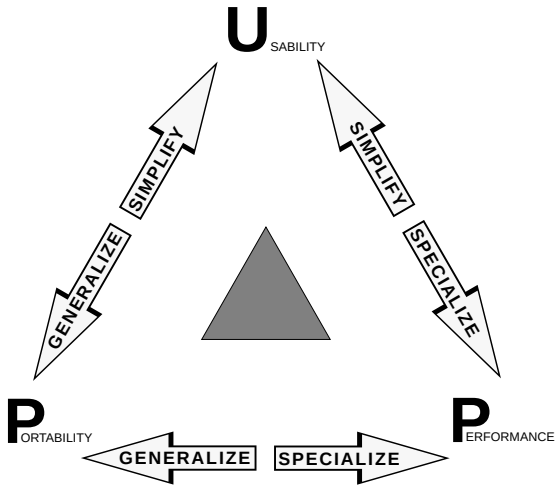



Figure 1.1: The PUP Triangle depicts the conflicting relationships of performance, usability, and portability. Each arrow is labeled with a description of the principal means for achieving the property to which it points. Opposing arrows denote conflicting means.

parallel programming [SOHL+98]; it *is* portable in the sense that an application will run correctly on any machine with minimal modification. MPI provides the union of all major communication mechanisms; thus, a programmer may select the best ones for a particular machine to achieve good performance on that machine. Unfortunately, this kind of specialization prohibits portable performance, as argued above. Next, a principal benefit of High Performance Fortran (HPF) is that all legal Fortran programs are also legal HPF programs, a relationship intended to make the language more usable [Hig97]. Unfortunately, a particular HPF compiler may or may not generate an efficient parallel implementation of the Fortran code; thus, the language rates poorly on both performance and portability.

### *1.2 An Approach to Practical Parallel Programming*

We address these impediments to practical parallel programming in the integrative design of language-level abstractions, compiler technology, and predictive performance models. Roughly speaking, language-level abstractions address the usability goal of simplification. With well designed abstractions, the programmer is presented with a computational view that hides complex details. It is then the role of the compiler to map these abstractions to a machine, specializing them

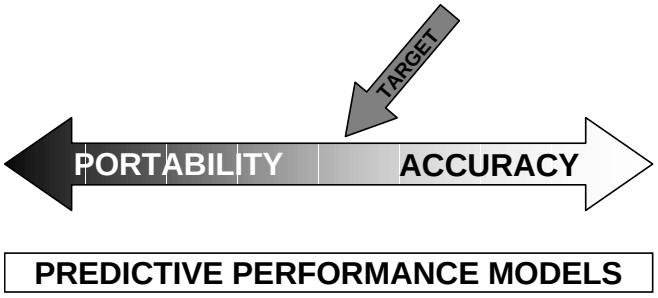



Figure 1.2: Depiction of the relationship between the accuracy and portability of predictive performance models.

to a particular architecture. The tension between performance and usability persists; thus, the design of abstractions and compiler technology must be guided by a portable predictive performance model.

Predictive performance models play an important role in practical parallel programming, as demonstrated in related work [Ngo97, CCL+98b]. A predictive performance model permits programmers to reason about the expected performance behavior of their codes, guiding them to efficient solutions. Nevertheless, there is a hazard to predictive performance models. As one becomes more certain of exactly how a code will perform on a particular machine, the prediction becomes increasingly dependent on the properties of that machine, as typified by the LogP model [CKP+93]. Figure 1.2 illustrates this phenomenon. Accuracy and portability are the end points of a continuum, and the goal is to find language features and abstractions that predict performance only to the degree necessary, maintaining a balance between predictability and portability. Only then will a predictive performance model be effective, independent of the target machine (*i.e.*, portable).

How do we decide the "degree necessary"? The work in this dissertation is guided by the two principal abstract properties defined by the CTA parallel machine model [Sny86]: parallelism and locality. Clearly, parallelism is essential in exploiting parallel machines. Equally important is locality, for all large-scale parallel computers consist of multiple processing elements separated by a relatively slow—as compared to processor performance—network. Locality minimizes dependence on the network, reducing overhead that is not fundamental to the computation.[1] A language that

[1] Although programming languages and computer architectures have been proposed to hide network latency, eliminating the significance of locality, no such system has yet conclusively obviated locality [ACC+90, Val90]

permits a programmer to reason about a code's parallelism and locality is well equipped to provide a predictive performance model that is both accurate and portable.

Just as fundamental parallel properties should be made apparent to programmers, secondary details must be abstracted in support of usability. Tedious, yet tractable, details of parallel programming should be relegated to the compiler, just as in the C programming language where details such as register allocation and instruction scheduling are left to a compiler. Thus, the programmer and the compiler have specific and distinct roles.

- The *programmer* is responsible for abstract parallel and sequential algorithmic issues.
- The *compiler* manages the tractable elements of mapping abstract representations to a particular machine.

We call the division of labor between programmer and compiler *programmer-compiler separation*, for it determines the responsibilities of each entity. In our division, a portable predictive performance model guides the design of abstractions and compiler technology, defining the roles of the programmer and compiler. A key property of this division is that the compiler is responsible for "tractable elements…" This stands in stark contrast to the large body of research in automatic parallelization, which advocates complete parallelization by compiler. The jury is still out on the efficacy of automatic parallelization, but a practical system has yet to be constructed despite more than two decades of effort. This dissertation and prior work argue that automatic parallelism is infeasible, for in the general case a compiler must discover new parallel algorithms from sequential representations, which is not manifestly tractable.

Nevertheless, the compiler's role is *not* trivial. This dissertation demonstrates that despite relieving the compiler of discovering parallelization, it must still perform sophisticated analyses and transformations in order to realize efficient and portable parallel code.

### *1.3 Practical Parallel Programming with Advanced–ZPL*

The Advanced–ZPL programming language (A–ZPL) embodies the above approach to practical parallel programming. A–ZPL is an array-based language designed from first principles to solve data-parallel scientific and engineering problems [Sny99]. A–ZPL (the successor to ZPL) has been in public release since 1997 and has seen production use by applied mathematicians, astronomers,

civil engineers, computer scientists, and physicists, among others. The A–ZPL language abstractions permit programmers to reason about the parallel and locality implications of their code, for the abstractions are subject to a portable predictive performance model. At the same time they are sufficiently abstract and intuitive that the language is easy to use as compared to the alternatives.

This dissertation examines representative portions of the A–ZPL language and its compiler and evaluates them in the light of programmer-compiler separation guided by a portable predictive performance model. We explore elements of the language that demonstrate the role of programmer-compiler separation. On one hand, we examine a complex task that is entirely managed by the compiler: the generation of loop nests from array statements. On the other hand, we consider a circumstance where new language abstractions dramatically improve the efficacy of language: support for pipelining wavefront computations. In the process, we illustrate how A–ZPL simultaneously achieves performance, usability, and portability.

Loop generation is a performance-critical aspect of array language compilation. Effective loop generation in A–ZPL consists of two components, statement fusion and array contraction. *Statement fusion* is analogous to loop fusion except that it is performed by the compiler on array statements before they have been converted to scalar loop nests. *Array contraction* is a program transformation enabled by statement fusion that permits a single scalar value to be used in place of an array. Together, these optimizations dramatically improve data cache behavior and performance—frequently by 25% and in some cases by a factor of 10—in most programs. In addition, array contraction reduces memory consumption, permitting larger problems to be solved in a fixed-sized memory. We argue that the task of loop generation is eminently tractable for the compiler, and our experiments support this claim.

Wavefront computations occur frequently in scientific programs, for example in solvers and dynamic programming codes. We show that fully automatic approaches to parallelizing such codes are impractical. We describe the design and implementation of a general language abstraction that consistently admits efficient pipelined parallel implementations of wavefront computations, and we show that this approach consistently performs better than the alternatives.

This dissertation is an in-depth study of the development of language-level abstractions and compiler technology guided by portable predictive performance models. All models are well defined, and all abstractions and compiler techniques are implemented and evaluated in the context of

a complete, practical, parallel programming language.

### *1.4 Contributions*

The work in this dissertation makes the following contributions.

- It uses the A–ZPL parallel programming language to illustrate the value of programmer-compiler separation guided by a portable predictive performance model.
- It motivates and justifies particular A–ZPL language design decisions in support of programmer-compiler separation considerations.
- It presents and experimentally evaluates a unique technique for array contraction.
- It introduces and evaluates a novel language abstraction for representing wavefront computations for pipelined parallel execution.
- It quantitatively and qualitatively evaluates the role of array contraction and the new abstractions in the design of a significant application.
- And it describes core elements of the A–ZPL compilation process, including efficient loop generation for array statements.

### *1.5 Thesis Outline*

The remainder of this document is structured as follows. The next chapter gives an overview of the A–ZPL programming language, and Chapter 3 presents related work and background, including discussions of predictive performance models, the A–ZPL performance model, our A–ZPL compiler, and A–ZPL's genesis. Chapter 4 describes and evaluates array statement fusion and array contraction in the A–ZPL compiler. Chapter 5 presents an A–ZPL language abstraction for the unambiguously parallel representation of pipelined parallel wavefront computations. Chapter 6 is a case study of the ASCI SWEEP3D benchmark [Accb], serving as a complete description of developing an entire A–ZPL application that benefits from the work in Chapters 4 and 5 as well as the rest of the A–ZPL system. The final chapter gives conclusions and proposes future work.

# Chapter 2

# THE A–ZPL LANGUAGE

This work has been performed in the context of the Advanced–ZPL (A–ZPL) parallel programming language, developed by the author and colleagues beginning in 1993 and continuing to the present time. This chapter introduces the basic concepts of the A–ZPL language. Only those facets of the language pertinent to this document are introduced, but complete language descriptions appear in the literature [Sny99, CCL+00].

The simple, yet complete, A–ZPL program that appears in Figure 2.1 serves as a running example in this chapter. As is evident from the sample code, A–ZPL has many of the same data types—boolean, integer, floating point, record, array—and control structures—**`if`**, **`for`**, **`while`**, **`repeat`**—found in standard imperative languages such as C and Pascal. In addition, it includes a typical assortment of arithmetic and logical operators.

Rather than reiterate these commonalities, this chapter highlights the distinguishing facets of A–ZPL. The following sections discuss the concepts of regions, region operators, array operators, and overall program structure; and the final section summarizes.

## *2.1 Regions*

The fundamental concept of the language is the region, which encapsulates parallelism, describes data distribution, and provides concise and readable syntax [CLLS99]. A *region* is an index set. For example, the following declaration defines a region `R` that includes the indices in the set $\{ (1,1), (1,2), \ldots (1,n) \ldots (n,1), (n,2), \ldots (n,n)\}$.

```
region R = [1..n,1..n];
```

As shown on line 6 of Figure 2.1, the region bounds can be more involved. Regions may have any static rank, and the upper and lower bounds of each dimension must be integral values. Mechanisms for specifying strided, hierarchical, and sparse regions are described elsewhere [Sny99, CDS00,

```
 1  program thinner;
 2
 3  config var m: integer = 10;             -- runtime constants
 4             n: integer = 20;
 5
 6  region R = [-m/2..+m/2,-n/2..+n/2];     -- declarations
 7
 8  direction north = [-1, 0];    east  = [ 0,+1];
 9            south = [+1, 0];    west  = [ 0,-1];
10
11  procedure skeletonize(var S: [R] integer);
12  var Obj, T, Temp: [R] integer;
13      err: integer;
14  [R] begin
15  [east of  R] S := 0;                     -- initialize boundary conditions
16  [west of  R] S := 0;
17  [north of R] S := 0;
18  [south of R] S := 0;
19
20  [R]   Obj := S;                          -- copy input image
21        repeat                             -- iterate over the thinning algorithm
22          Temp := min(min(S@north,S@east),min(S@south,S@west));
23          T := Obj + Temp;
24          err := max<< abs(S-T);
25          S := T;
26        until err = 0;
27
28        S := (S>=S@north) & (S>=S@east) & (S>=S@south) & (S>=S@west) & (S!=0);
29  end;
30
31  procedure thinner();                     -- entry procedure
32  var S: [R] integer;
33      objfile: file;
34  [R] begin
35        objfile := open("object.dat", "r");
36        read(objfile, S);
37        close(objfile);
38        skeletonize(S);
39        writeln("%d ":S);
40  end;
```

Figure 2.1: Complete, sample A–ZPL program. This program computes the shape skeleton (medial axis) of a two dimensional object that is represented as non-zero pixels. The algorithm takes as input a discretized representation of an object and iteratively performs *volume thinning* [BB82] by nearest neighbor minimization until only the skeleton remains.

Cha00, CLS98]. Once defined, regions are used to declare arrays, specifying the size and shape of these arrays. The following code fragment from line 12 of Figure 2.1 declares `Obj`, `T`, and `Temp` to be two-dimensional integer-typed arrays with memory allocated for each index in region `R`.

```
var Obj, T, Temp: [R] integer;
```

Regions are also used to specify indices for array operations. For example, line 20 of the figure shows how array assignment generalizes scalar assignment.

```
[R] Obj := S;
```

Elements of the right-hand side, whose indices are given by the region `R`, are assigned to their corresponding elements on the left-hand side, whose indices are again given by the same region. Here, we say that Line 20 is in the *scope* of region `R` because `R` is attached directly to the statement, and hence both the `Obj` and `S` arrays use the same region. In general, a $d$-dimensional region defines indices for all $d$-dimensional array expressions in its scope. Regions also apply to compound statements, so Line 14 defines a scope that applies to the entire body of the procedure, except for those that are within the scope of more deeply nested regions. Finally, regions are dynamically scoped, so procedures can inherit regions from the scope of their call site. As described below, other constructs, such as the reduction operator, require two regions, while expressions involving only scalar variables require none.

Regions are often named for convenience and clarity, but this is not required. For example, the following lines might be used to assign zeroes to the upper triangular portions of an $n \times n$ array.

```
          for i := 1 to n do
[i,i..n]    A := 0;
          end;
```

In specifying a region, elided dimensions are inherited from the dynamically enclosing region scope. For example, `[i,]` refers to row `i` of the enclosing region.

A slightly more complicated array statement is given on Line 23,

```
T := Obj + Temp;
```

which performs an array addition and then assigns the corresponding sums to elements of `T`. Array addition generalizes the scalar `+` operator by summing corresponding elements of its array operands, so the result of evaluating `Obj+Temp` is an array of sums. In general, any scalar operation or procedure may be used with scalar arguments in this way, as seen in Line 22's application of the scalar `min` function to array operands. This is called *functional promotion.* Similarly, when a scalar is used in a context where an array is expected, the scalar implicitly becomes the rank and size of the expected array, as exemplified by the assignment of scalar 0 into the east boundary of array `S` in line 15. This is called *scalar promotion.*

```
east  = [ 0, 1];
north = [-1, 0];
ne    = [-1, 1];
east2 = [ 0, 2];
```

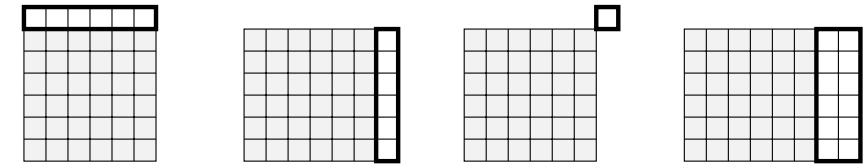



Figure 2.2: Examples of the `of` region operator. Shading represents region `R`, and the outlined area represents the region specified beneath each example.

### *2.2 Region Operators*

To simplify the specification of regions, A–ZPL provides a number of operators that describe new regions relative to existing ones. Regions are often constructed using *directions*, which are user-defined vectors. Line 8 shows the definition of two directions, which are used to describe the geometric relationships used in this program.

```
direction north = [-1, 0];    east  = [ 0,+1];
```

Once defined, directions can be used with the various region operators such as the **`of`**, **`in`**, and **`at`** operators, here informally defined. The **`of`** operator applies a vector to a region of the same rank and produces a new region that is adjacent to the original region in the direction of the vector. For example in Figure 2.1, `[north of R]` refers to the row of indices above `R`. Figure 2.2 shows examples of how the magnitude and signs of the direction vector determine the size and placement of the resulting region.

Lines 15–18 of the example show how boundary conditions are defined. Two points are noteworthy. First, A–ZPL simplifies memory allocation by implicitly defining storage for boundary values, which are defined to be any portion of an array that is initialized in the scope of a region that uses an **`of`** operator. Thus, array `S` is allocated sufficient memory to store elements in `[east of R]`, `[west of R]`, `[north of R]`, and `[south of R]` despite the fact that it is declared using region `R`. The second point is that boundary conditions are typically difficult to deal with in scientific computations because they make it impossible to treat the entire computational space uniformly. A–ZPL provides support for boundary conditions through the use of the `wrap` and `reflect` statements, which can be used to initialize periodic and mirrored boundary conditions. `Wrap` and `reflect` are described elsewhere [Sny99]. Furthermore, the use of regions allows boundary condition code to be

```
east  = [ 0, 1];
north = [-1, 0];
ne    = [-1, 1];
```

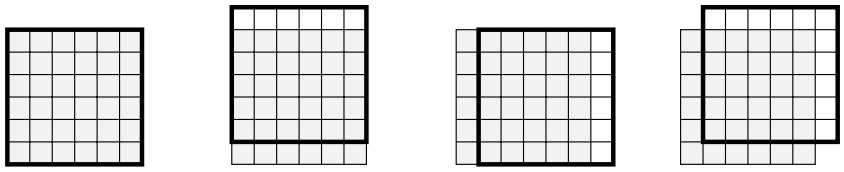



Figure 2.3: Examples of the `at` region operator. Shading represents region `R`, and the outlined area represents the region beneath each example.

clearly separated and identified, as evident in Figure 2.1.

The **`in`** operator is the dual of the **`of`** operator, producing a region that is *inside* the original region and adjacent to the border in the direction of the vector. For example, `[north in R]` refers to the first row of indices inside of region `R`.

Finally, the **`at`** operator translates an index set by some specified vector without changing the size or shape of the original region. For example, the region `[R at north]`refers to the translation of the `R` region by one in the `north` direction. In general, the **`at`** operator produces a new region by adding the specified direction to each index of the specified region. Examples are given in Figure 2.3.

### *2.3 Array Operators*

Regions govern the referenced indices for entire statements, but array operators adjust the indices for individual array references. The **`at`** region operator has an alternate form, `@`, that can be applied to individual arrays, the only region operator for which this is true. For example, the statement on line 22 computes, for each point, the minimum of its four neighboring elements.

```
Temp := min(min(S@north,S@east),min(S@south,S@west));
```

The right-hand side of this statement first uses the built-in `min` function to compare corresponding elements of array `S`'s north and east neighbors. Here, correspondence is natural; for example, the upper left-hand elements of arrays `S@north` and `S@east` correspond. The result of this function evaluation is an array whose values are the minimum between `S`'s `north` and `east` neighbors. The remainder of the right-hand side operates analogously, minimizing over the `south` and `west`

arrays, and then combining these results into a single array of minimum values.[1] As mentioned earlier, the `min` function is a scalar C function that is promoted to operate on each element of an array expression.

In addition to providing mechanisms for simplifying the specification of array indices, A–ZPL provides a number of array operators, such as reduce and scan operators, that simplify the manipulation of arrays. The reduce operators perform an associative, commutative operation, such as + or *max*, on all elements of an array, producing a single scalar. Line 24 shows an example of the max-reduce operator, which takes the largest element of the absolute values of `S-T` and assigns the result to the scalar variable `err`.

```
err := max<< abs(S-T);
```

The scan operators apply an associative, commutative operator to an array of values. The resulting value is an array in which each element is the reduction of all preceding elements of the array. Scan operations are also known in the literature as the parallel prefix [LF80, Ble90]. The syntax of a scan operator is shown below, where we assume that `A` and `B` have been declared to be of the same rank.

```
A := max|| B;
```

The syntax of the reduce, scan and flood operators (defined below) are chosen to remind users that reduce (**<<**) produces a smaller result, scan (**||**) produces a result of the same size and shape, and flood (**>>**) produces a larger result.

Reductions collapse all dimensions of an array to a single scalar, while scans perform the parallel prefix operation across all dimensions of an array. A–ZPL provides variations of these operations, known as partial reductions and partial scans, which operate over a subset of the dimensions. For example, the following declarations and statement perform a partial reduction of the `B` array and assign the result to the array `A`.

```
region I  = [1..n,1   ];
       IJ = [1..n,1..n];

[I] A := +<<[IJ] B;
```

[1] Lines 22 and 23 would more naturally be combined into a single statement, eliminating the use of the `Temp` array, but the statement was split for pedagogical reasons. Nevertheless, the compiler eliminates such temporaries as described in Chapter 4.

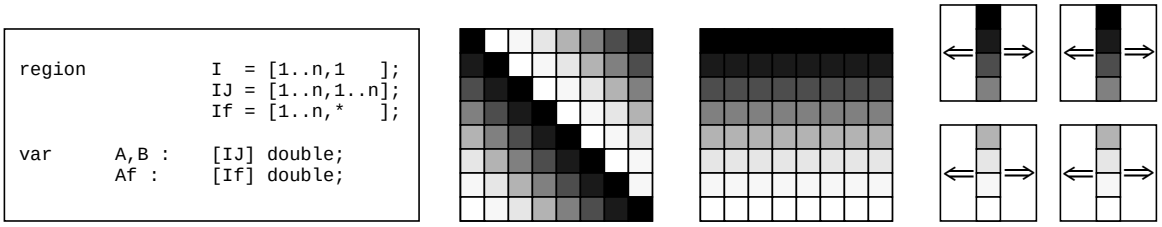



Figure 2.4: Illustration of flooding. (a) and (b) give declarations and initial value of array `B`. (c) and (d) show effects of flooding. In (d), the target array is distributed across a $2 \times 2$ processor grid and it's second dimension is declared floodable, so only the defining values are represented on each processor.

The source region, `IJ`, applies to the `B` array, and the destination region `I` applies to the `A` array. Here, the region names are chosen to indicate that the region `IJ` includes all indices in a three-dimensional region, while the region `I` includes a plane of the same region. Thus, the source and destination regions combine to indicate that the second dimension is the collapsed dimension.

The flood operator (**`>>`**) performs the dual of reduction, replicating a slice of an array. Like partial reductions, the flood operator encodes a region, specifying the portion of the source array to be replicated. As always, the enclosing region specifies the target indices. For example—assuming the declarations given in Figure 2.4(a)—Figures 2.4(b) and (c) illustrate the replication of the first column (defined by region `I`) of array `B` by flooding. Again, the source and destination regions combine to indicate that the replication is across the second dimension. In order to conserve memory, an array dimension may be declared as flooded (via the token `*`), indicating that all data along that dimension will have only a single value. Storage is only allocated for the defining values, as illustrated in Figure 2.4(d), but the array can be referenced as if it is a normal array.

It is often convenient to replicate data dimensionally, like flood, but only on a per-processor basis. Thus, an array dimension may be declared a region-grid, or r-grid, via the token `::`. For example, the following declaration produces a row vector unique to each processor row.

```
var X : [::,1..n];
```

The most general of A–ZPL's array operators is the remap operator (`#`), which permits arbitrary array-level indexing of array data. For example, consider the following statement, where `A`, `B`, `I`,

and `J` are two-dimensional arrays.

```
[R] A := B#[I,J];
```

The right operand (`[I,J]` in this case) is a sequence of integer arrays that are used to index into the left operand (`B`). In this case, arrays `I` and `J` define the source indices for the first and second dimensions, respectively, *gathering* elements from array `B` to array `A`. When a remap operator appears on the left-hand side of an assignment, a *scatter* is performed. Assignment operators such as `+=` or `*=` are used to accumulate multiple values that scatter to the same index. Otherwise, an arbitrary value persists when there is a conflict. For example, if integer arrays `I` and `J` containing only ones, the following statement has the effect of assigning the product of the region `R` elements of `B` to index $(1, 1)$ of `A`.

```
[R] A#[I,J] *= B;
```

The remap operator can be used to perform a matrix transpose as follows.

```
[R] A := B#[Index2,Index1];
```

Here, `Index1` and `Index2` are built-in A–ZPL arrays whose elements contain their row index and column index, respectively. These arrays are defined to be conformable with any array, and are generalized to all dimensions that appear in a program.

### *2.4 Overall Program Structure*

The overall structure of an A–ZPL program is apparent in Figure 2.1. The name of the program is given on the first line. This name is used to define the main procedure, where program execution will begin. The top of the program contains various declarations, and this is followed by a series of procedure definitions.

The first set of declarations specify configuration variables, which are variables whose values are bound once at load time and remain constant thereafter. Configuration variables are useful for specifying values of parameters that are likely to be specific to a given instantiation of a program. These variables can be assigned default values as shown on Lines 3–4, assigned values on the command line, or initialized through configuration files that are specified on the command line. The next declarations define regions in terms of the configuration variables, and these are followed by

the declaration of directions. This program has no global variables, but globals can be defined if necessary.

The main procedure, `thinner()`, declares some local variables, performs I/O to initialize data, and then performs the actual thinning by invoking the `skeletonize()` procedure. The `skeletonize()` procedure illustrates how array parameters may be declared with specific regions defining their size and shape. By contrast, the **`read`**`()` and **`writeln`**`()` routines can operate on arrays of arbitrary size and shape because they were defined without providing specific regions for their array parameters. Thus, line 34 attaches regions to the call sites of lines 36 and 39, and the dynamic scoping of regions propagates these regions to the bodies of the I/O routines.

### *2.5 Summary*

A–ZPL is the parallel programming language upon which the work in this dissertation builds. It is an array language principally distinguished by its use of index sets called regions, which are used to define arrays and specify the extent of a statements computation. Individual dimensions of a region may be classified as flooded or r-grid, indicating single coherent or non-coherent values along that dimension. A number of region operators are available to simplify the construction of regions, particularly those at the boundaries. Array operators permit the modification of the indices for a particular array reference and algebraic operations, such as reduction. The large number of applications written in A–ZPL suggests that its design well covers the needs of the data-parallel domain.

# Chapter 3

# BACKGROUND AND RELATED WORK

High performance parallel computing is a broad discipline with an extensive history. This chapter provides appropriate background and related work, setting the stage for the remainder of this dissertation. The next two sections give the general context and summarize related parallel programming systems. Section 3.3 discusses performance models, and the final two sections outline the structure of the University of Washington A–ZPL compiler and give historical context.

## *3.1 Context*

This work is conducted in the context of high performance parallel computing, where it is necessary to achieve and sustain TeraFLOPS-scale performance. Today, performance of this magnitude can only come from a high degree of parallelism, so it is essential that parallel machines intended for this domain scale to potentially large numbers of processors.

Furthermore, this work concentrates on the *data parallel* form of parallelism. Data parallelism arises when the same or a similar operation is applied to each element of a data set. Often, these operations can be performed in parallel. Matrix and array-based computations are frequently data parallel. Data parallelism is to be distinguished from *task* or *functional parallelism*, in which parallelism exists between portions of a computation performing different operations. Data parallel computations are very common in science and engineering problems, because they often exploit very large data sets over which computation is only practical with parallelism.

There exists a rough association between data and task parallelism and *SIMD (single-instruction, multiple-data)* and *MIMD (multiple-instruction, multiple-data)* parallel machines, respectively. SIMD machines are designed to perform the same operation (a single instruction) on multiple data elements, so they are well equipped to exploit data parallelism. On the other hand, they can not exploit task parallelism. MIMD machines are designed to perform different operations on multiple

data elements; thus, they can perform either data or task parallel computations. Economic concerns have made SIMD machines infeasible, to the point that only MIMD machines exist today. Despite the relationship between data parallelism and SIMD machines, A–ZPL effectively exploits MIMD machines and it provides features that support limited forms of task parallelism.

### *3.2 Related Parallel Programming Systems*

Perhaps the best known language effort for parallel computing is High Performance Fortran (HPF) [Hig97]. HPF was designed by extending the sequential Fortran 90 language to support the distribution of arrays across multiple processors, resulting in parallel computation. Programmers may give suggestions for array alignment and distribution in the form of directives, though their use is optional and they may be ignored by the compiler. This flexibility in implementation has two drastic effects: *(i)* programmers have no direct means for determining the communication overheads associated with their programs since communication is dependent on data distribution and alignment, and *(ii)* compilers are free to distribute data as they see fit, implying that a program which has been tuned to work well on one platform may perform terribly when compiled on another system. This lack of a performance model in the language is completely antithetical to the notion of portable performance.

Ngo *et al.* demonstrate that HPF's failure to specify a performance model results in erratic execution times when compiling HPF programs with different compilers on the IBM SP-2 [NSC97]. To alleviate this problem, tools such as the dPablo toolkit [AWMC+95] have been designed to give source-level feedback about compilation decisions and program execution. However, these tools are tightly coupled to a compiler's individual compilation model, and therefore they do not directly aid in the development of portable programs.

NESL [Ble92] is a parallel functional programming language. Its designers recognized that in the parallel realm the ability to reason about a program's execution is crucial, so a work/depth-based performance model was designed to support this endeavor [Ble96]. Although this model well matches NESL's functional style and allows for coarse-grained implementation decisions, it uses a very abstract machine model that reveals little about the mapping of NESL constructs to actual architectures. For example, the cost of interprocessor communication is considered negligible in the

NESL model and is therefore ignored entirely.

C∗ [Thi91] is an extension to the C programming language that was developed for programming the Connection Machine. Several aspects of its design do an excellent job of making the mapping of C∗ programs to the hardware transparent. For example, the Connection Machine architecture supports two general types of interprocessor communication with significantly different overheads—*grid communication* and the more costly *general communication*. This disparity is reflected in the language by its syntactic classification of array references as being either grid or general. Although this does an excellent service for the Connection Machine programmer, its benefits are diminished when C∗ is implemented on different architectures since they may support additional forms of communication with intermediate costs, *e.g.*, broadcasts along sub-dimensions of a processor grid.

As an alternative to parallel languages, many runtime libraries have been developed to support the creation of portable parallel codes. As libraries, these approaches do not offer the same syntactic benefits as languages, such as A–ZPL, and they cannot benefit from the same compiler optimizations that a language does.

The most notable libraries are those that provide support for message passing, such the Parallel Virtual Machine (PVM) [BD94] and the Message Passing Interface (MPI) [SOHL+98]. These libraries have been hailed as successes due to their widespread implementation on numerous parallel and sequential architectures, and for the relative ease with which codes written on one architecture can be run on another. However, the libraries are not without their drawbacks. First, they put the entire burden of parallel programming on the users, requiring them to code at a per-processor level and manage all memory and communication explicitly. This is tedious and error prone, and is considered by many to be equivalent to programming sequential computers in assembly language. In addition, the libraries restrict the user to a particular paradigm of communication, which may or may not be optimal for a given architecture [CCS98]. Although extensions to the libraries [SOHL+98] seek to alleviate this problem by supporting a richer set of communication styles, this does not solve the problem because to achieve optimal performance, a program would have to be rewritten for each machine to use the interface that is most appropriate.

LPARX [KB94] is a library that supports the parallel implementation of non-uniform problems. LPARX provides user-controlled index sets and a more general version of A–ZPL's regions that support set theoretic operations, such as union, intersection, and difference. LPARX programmers

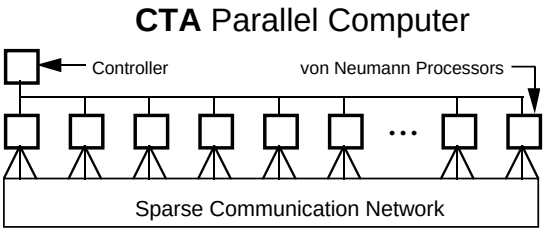



Figure 3.1: The CTA abstract parallel machine model.

can specify the distribution of index sets to processors, relying on the runtime system to implement transparent interprocessor communication for non-local array references. For this reason, LPARX does not provide portable performance.

HPC++ [JGB97] extends C++ by providing class libraries to support both task and data parallelism. HPC++ uses a parallel implementation of the C++ Standard Template Library to provide parallel container classes and parallel iterators, and HPC++ uses pragmas to identify parallel loops. HPC++ also provides support for multi-threaded programming. In short, HPC++ supports task parallelism and a wider range of data structures via lower-level mechanisms than those in A–ZPL.

In summary, A–ZPL can be distinguished from all these systems in that it provides portable performance, a direct result of its portable performance model.

## *3.3 Modeling Parallelism*

### *3.3.1 Background*

A machine model captures the salient characteristics of a machine so that its users are presented with an abstract and more manageable view of its operation. For example, the so-called von Neumann machine model captures the salient characteristics of modern sequential computers: a stored program, a common memory for data and code (accessible in unit time), and a CPU to perform arithmetic and logical operations [vN45]. That a single model describes virtually all serial machines contributes directly to the portability of software. This model has persisted since the 1940s, though data caches and instruction level parallelism are threatening its veracity.

The CTA[1] machine model is intended to serve as the parallel analog of the von Neumann machine model [Sny86]. A CTA machine (Figure 3.1) contains multiple von Neumann processors, each with a local memory. Processors are connected via a sparse network of unspecified topology. The latency of communication across the network is unspecified, but it is assumed to be significantly larger than the time to perform a local memory reference. All processors are connected via a *thin* channel to a controller that manages global synchronization operations. Thus, this model abstracts the performance characteristics of parallelism and locality, without the machine dependence that comes with undue specificity.

Machine models are too low-level for convenient programmer manipulation. The *programming model* builds abstractions upon the machine model. For example, in sequential programming, the imperative-procedural programming model [PZ96], typified by C or Fortran, provides symbolic naming, data structures, procedures, parameters, recursion, control structures, *etc*. These are useful abstractions that are not provided by the machine model directly but are important for practical programming.

The *phase abstractions* programming model is a parallel analog of the imperative-procedural programming model [Sny90, AGNS90], providing equivalent features as well as data allocation and processor assignment information. The programming model defines a scalable unit of parallelism that encapsulates three aspects of parallel computations—code, data and communication—so that performance-critical characteristics of a parallel program can be adjusted for different target machines. For example, the model allows the granularity of parallelism to be easily adjusted. By expressing the code, data and communication topology of parallel programs as independent units, the model encourages component reuse and simplifies the tuning process. Most significantly, the model does not obscure the underlying abstract parallel machine, the CTA [Sny93].

A–ZPL builds upon the CTA machine model and the phase abstractions programming model. Corresponding serial and parallel machine and programming models appear in Table 3.1.

Before we describe the A–ZPL performance model, we summarize some other parallel machine models. The PRAM (parallel random-access machine) models a parallel machine with $p$ serial processors that have “unit-time” access to a single shared memory [FW78]. This model ob-

[1]CTA is an acronym for candidate type architecture. A type architecture and machine model are synonymous.

Table 3.1: Serial and parallel models.

| | Serial | Parallel |
|---|---|---|
| Machine model | von Neumann | CTA |
| Programming model | Imperative-procedural | Phase Abstractions |
| Language | C, Fortran, Pascal, *etc*. | A–ZPL |

viously ignores the cost of communication between processors, which has yet—and is unlikely—to be realized in hardware. In fact, with the PRAM, absurd conclusions may be drawn, including constant-time sorting algorithms [Akl89]. Although the PRAM is useful in studies of the limits of parallelism, it is not a practical tool for guiding algorithm design.

The LogP model contains $P$ serial processors, each with a local memory, connected via a network of unspecified network topology. Rather than a unit-cost memory access, the performance characteristics are defined by four parameters: $L$ (latency for interprocessor communication), $o$ (overhead or amount of time a processor is engaged in a transmission), $g$ (gap or minimum time between consecutive communication operations), and $P$ (number of processors). The LogP model has been used to predict performance with great accuracy [DCSM96, ABLZ99], but it is precisely this accuracy that impedes portability.

The following section concretizes the above discussion, describing how A–ZPL programmers reason about the expected performance of programs.

### *3.3.2 A–ZPL Performance Model*

As with most array languages, the semantics of array assignment are that the right-hand side is evaluated before it is assigned to the left-hand side, and one array statement is logically completed before the subsequent statement is executed. Each array statement specifies a collection of operations on the elements of the statement's arrays. This collection can logically be performed in any order, which allows the implementation to execute the operations in parallel. Thus, the amount of parallelism in an A–ZPL program is described by the region attached to each statement. At the same time, these array language semantics allow programmers to reason about A–ZPL programs as

if they were sequential programs.

To achieve parallelism, arrays are distributed across processors based on the distribution of their defining regions. The distribution of regions is restricted to obey the invariant that *interacting regions* are distributed in a *grid-aligned* fashion.

Two regions are considered to be interacting when they are both explicitly or implicitly referenced in a single statement. Explicit region references are those encoded in array operators (*e.g.*, partial scans and reductions) and those that specify the indices of an array statement. Implicit region references are those that are used to declare the arrays appearing in the statement. For example, the following statement (Figure 2.1 line 18) implicitly references region `R` because array `S` is declared over `R` and explicitly references region `south of R`, so they are interacting regions.

```
[south of R] S := 0;
```

Grid-aligned means that if two $n$-dimensional regions are partitioned across a logical $n$-dimensional processor grid, both regions' slices with index $i$ in dimension $d$ will be mapped to the same processor grid slice $p$ in dimension $d$. For example, since `R` and `north of R` are interacting, they must be grid-aligned, and therefore column $i$ of `north of R` must be distributed across the same processor column as column $i$ of `R`. Moreover, grid-alignment implies that element $(i, j)$ of two interacting regions will be located on the same processor. This is a key property of our distribution scheme. Note that using a blocked, cyclic, or block-cyclic partitioning scheme for the indices of a set of interacting regions causes the regions to be grid-aligned. Our A–ZPL compiler uses a blocked partitioning scheme by default, and for simplicity we will assume this scheme for the remainder of this dissertation.

Once regions are partitioned among the processors, each array is allocated using the same distribution as its defining region. Array operations are computed on the processors containing the elements in the relevant region scopes.

Grid-alignment allows A–ZPL to provide syntactic cues to indicate where the compiler will generate various forms of communication. In particular, @'s reductions and scans may involve indices that are distributed to different processors. For example, the statement

```
T := Obj + min(min(S@north,S@east),min(S@south,S@west));
```

refers to both `T` and `S@north`, and the indices that are not common to both regions may belong to different processors.

One final characteristic of A–ZPL's data distribution scheme is that sequential variables are replicated across processors. Coherency is maintained through redundant computation when possible, or interprocessor communication when not. This coherency-related communication can only be induced by a small number of operations, such as reductions and scalar **`read`** operations, so this type of communication, as with all communication in A–ZPL, is apparent to the programmer.

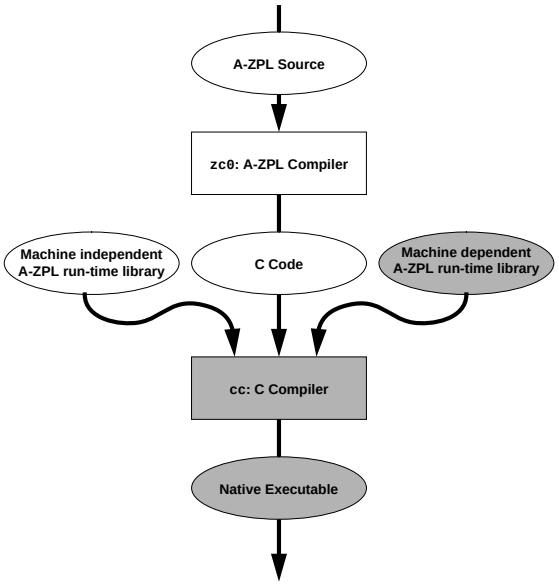
## *3.4 UW A–ZPL Compiler and Run-time System Overview*

To date, the University of Washington A–ZPL compiler is the only A–ZPL compiler in existence. It is a multi-pass, optimizing, source-to-source translator that produces ANSI C code that is compiled with a machine's native compiler—here called the back-end compiler—and linked with machine-dependent and independent run-time libraries. Figure 3.2(a) summarizes the process. Note that shading indicates an item is customized to the target machine. The component labelled "A–ZPL Compiler" is the subject of this section. Figure 3.2(b) summarizes its operation, and Sections 3.4.1 – 3.4.4 elaborate. This discussion is limited to the details of the UW A–ZPL compiler that distinguish it from compilers for other languages. We conclude with a summary of the status of the current implementation.

### *3.4.1 Structures*

All analyses and transformations are performed on an abstract syntax tree (AST) representation of an A–ZPL source program. An AST alone is a sufficient representation because the A–ZPL compiler only performs high-level transformations (as described below), leaving most scalar compilation tasks to the back-end C compiler. A control flow graph (CFG) is never constructed, for it is implicit in the AST. A call graph is used for interprocedural analyses.

An array statement dependence graph (ASDG) represents dependences between statements at the source level. The dependences that arise from the compiler generated loop nests are never explicitly constructed, for they are implicit in the ASDG where they are far more concisely represented. Chapter 4 develops this concept.




(a)

1. Parsing
2. Type Checking
3. Normalization
   - Temporary insertion
   - M-loop insertion
4. Communication Insertion
5. Optimization
   - Communication optimization
   - Statement fusion
   - Array contraction
   - Storage conflict elimination
   - Static memoization for stencils
   - Scalar optimizations
6. C Code Generation

(b)

Figure 3.2: Structure of A–ZPL compilation system. (a) The complete system. (b) The logical structure of the A–ZPL compiler. Shaded elements are machine-dependent.

### *3.4.2 Core Transformations*

The principal role of the A–ZPL compiler is to scalarize and parallelize array statements, precisely those tasks that the back-end C compiler does not perform. *Scalarization* is the process of transforming an array statement into a semantically equivalent scalar (*i.e.*, C) implementation. *Parallelization* is the distribution of arrays and the insertion of communication for parallel operation.

#### *Scalarization*

Array temporary insertion is the first step of scalarization. Temporary arrays are necessary to implement certain array statements on scalar architectures. Specifically, temporary arrays are inserted for the following (examples appear in Figure 3.3): *(i)* actual parameters when their type does not exactly match the formals (coercion takes place during copy), *(ii)* actual parameters that are call-by-

```
procedure foo(X : [R] integer) ...

var P : [R] record
              a : integer;
              b : integer;
            end;

var A,B : [R] integer;
var T : [R] integer;
```

```
foo(P.b);
```
...becomes...
```
T := P.b;
foo(T);
```
(a)

```
foo(A);
```
...becomes...
```
T := A;
foo(T);
```
(b)

```
A := 2 * +|| B;
```
...becomes...
```
T := +|| B;
A := 2 * T;
```
(c)

```
A := A@w/4;
```
...becomes...
```
T := A@w/4;
A := T;
```
(d)

Figure 3.3: Examples of temporary array insertion.

value, *(iii)* functions and language-level operators that return arrays, and *(iv)* right-hand-side array expressions that alias the left-hand-side array.

Note some of these array copies need only create a new array descriptor into the existing array data, as in example (a). In some cases the array temporary is unnecessary. For example, in example (b) the temporary is unnecessary if formal array parameter `X` is not modified in procedure `foo()`. This optimization can dramatically impact performance unlike in the scalar case. Example (d) can also be optimized, for a temporary is not required if the loop over the second dimension iterates from high to low indices. These temporary arrays are eliminated by array contraction, introduced in Chapter 4.

After temporary arrays are inserted, array statements are replaced with multi-loops, or m-loop. M-loops contain all the necessary information to generate a loop nest to implement a particular array statement or statements, without actually explicitly representing the loop nest. This is possible because array statement are implemented by a highly restricted class of loop nests. This compact

high-level internal representation greatly reduces compiler complexity without sacrificing any expressivity. M-loops are parameterized by the computation they represent and the structure of the loop nest (loop order, direction, tiling, peeling, unrolling, *etc*.). Optimization of loops due to array statements becomes a matter of manipulating properties of m-loops. An implication of not explicitly representing the control flow for implementing array statements is that basic blocks are large, exposing significant optimization opportunities even when only performing local analysis.

*Parallelization*

Distribution of arrays according to the regions that define them is described in Section 3.3. A–ZPL was designed so that interprocessor communication is only required for the array operators unique to A–ZPL (*e.g.*, `@`, `+<<`, *etc*.). Thus, the compiler introduces one or more library calls to implement the necessary communication for each array operator.

### *3.4.3 Optimization*

Naïvely generated code may be highly inefficient; thus, aggressive optimization is essential for achieving performance on par or better than hand-coded applications. The most important A–ZPL optimizations appear in Figure 3.2(b), and they are summarized below. This set of optimizations was chosen for the follow reasons: *(i)* they allow A–ZPL code to compete with hand-coded programs, and *(ii)* they may be uniquely performed in A–ZPL due to its high-level intermediate representation. This is not to say that they can not be performed in other language contexts. Rather, in other contexts they are of questionable value, for they likely optimize infrequently occurring or easily thwarted idioms. Conversely, A–ZPL provides language-level support for the constituents that they optimize. All these transformations optimize elements of the generated code over which the programmer has no control (*e.g.*, communication placement and loop nest structure); thus, it is particularly important that the compiler generate quality code. The A–ZPL compiler leaves most scalar optimizations (*e.g.*, register allocation, instruction scheduling, *etc*.) to the back-end C compiler.

**Communication Optimization.** The A–ZPL compiler aggressively optimizes interprocessor communication [Cho99], for naïve communication can significantly limit parallel performance. The compiler performs three well known transformations: redundancy elimination, combining,

and pipelining. The first two minimize the volume and number communication steps, respectively. Pipelining, achieves latency hiding by overlapping data transmission and computation. Though these transformations are well known, the A–ZPL context permits unique and consistently effective application [CS97].

**Array Contraction and Statement Fusion.** Array statement fusion is equivalent to loop fusion [Wol96], except that it transforms m-loops, which imply loop nests, rather than loop nests directly. Like loop fusion, statement fusion can improve data locality. It can also form loop nests such that certain array references may be contracted to scalar references, conserving memory and improving cache performance. Array statement fusion and contraction are the topic of Chapter 4. As the reader will learn, array contraction can improve performance by as much as an order of magnitude and typically 20 to 30% in already highly optimized codes.

**Storage Conflict Elimination.** A storage conflict exists when multiple memory references refer to the same portion of memory, but there is no flow of data from one to the other. These are called false data dependences, for one statement only depends on the other in that they share storage, not data values. False data dependences often prevent other transformations, such as statement fusion and array contraction. Storage conflict elimination introduces extra memory resources in order to eliminate the conflict, enabling other transformations [Low00]. The net result is that storage conflict elimination can improve performance by as much as a factor of two.

**Static Memoization for Stencil Computations.** Stencil computations are very common in data parallel computations and they are concisely represented in A–ZPL. Often stencils redundantly compute values in different loop iterations. For example, imagine a stencil that sums the eight neighbors for each element of a two-dimensional grid. Iteration $j$ of the inner loop computes the sums `A[i-1,j-1]+A[i+1,j-1]`, `A[i-1,j]+A[i+1,j]` and `A[i-1,j+1]+A[i+1,j+1]`. Similarly, iteration $j+1$ computes the sums `A[i-1,j]+A[i+1,j]`, `A[i-1,j+1]+A[i+1,j+1]` and `A[i-1,j+2]+A[i+1,j+2]`. Static memoization for stencil computations reuses partial results between iterations, reducing the number of floating point operations [DCS00]. This transformation has been shown to improve performance by 15% in applications such as NAS MG.

**Scalar Optimizations.** Out of necessity, the A–ZPL compiler performs a small number of scalar optimizations. These are necessary for the following reasons: *(i)* the free use of C pointers thwarts some analyses in the back-end compiler, *(ii)* our studies indicate that back-end C compilers do not reliably or effectively perform certain critical transformations, or *(iii)* the transformation requires high-level semantic information that the back-end C compiler is unlikely to derive. For example, the compiler performs tiling in order to balance parallelism and communication overhead in pipelining wavefront computations, the topic of Chapter 5. The compiler performs loop invariant code motion and tiling in order to improve the performance of flood array access; this is unique in that the transformation is based on the type of the references, not how they are referenced. In addition all array references are optimized to minimize index calculation, effectively performing loop invariant code motion and strength reduction. This last transformation alone improves the performance of most codes by approximately a factor of two.

### *3.4.4 Run-time System*

Arrays are represented at run-time via a descriptor representing the structure of the array and containing a pointer to a block of memory that represents the content of the array. This permits arrays to be first class, dynamic, and cheaply aliased.

The bulk of the run-time library is devoted to performing communication. A portable machine independent library, called the Ironman interface, has been defined, and all A–ZPL compiled programs target this machine independent library [CCS98]. The Ironman interface is unique in that it presents *collective semantics*; thus, each routine does not have a particular meaning, but when used together they have a meaningful interpretation. The Ironman interface has been implemented on top of a diverse collection of communication mechanisms, including the message passing interface (MPI), the parallel virtual machine (PVM), the Cray Shared Memory Access Library (Shmem), and shared memory.

### *3.4.5 Current Status*

As of Autumn 2000, every language feature and compiler transformation described in this document is fully implemented. The A–ZPL language definition is not yet complete, so it and its compiler are

evolving, while always maintaining backward compatibility. Despite development in an academic environment, the compiler is considered beta quality. It is publicly available for download and evaluation at the A–ZPL web site [ZPL] in binary form for all common parallel and networked machines. It is routinely used by non-computer scientists for production application development, and it has been used to derive numerous published research results [DLMW95, LLST95, RBS96, WGS00].

### *3.5 Historical Overview of the Development of A–ZPL*

A complete history of A–ZPL is premature, for the language is yet to be completed. Nevertheless, the A–ZPL project is sufficiently mature that its past is worth recording for fear of losing it and so that one may better understand how it came to be. We sketch the critical developments along the path to A–ZPL. It is far from complete and, without a doubt, (unintentionally) biased.

It was widely held in the late 1970s and early 1980s that improving performance by exploiting some form of parallelism was a winning idea. This period saw the advent of a great many parallel platforms and designs, including the Carnegie-Mellon University Cm$*$, the CRAY-1, the Connection Machine, Cosmic Cube, Cray X-MP, Intel iAPX 432, VAX 11/782, Denelcor HEP, and the NYU Ultracomputer. Experience with these machines demonstrated that it was difficult to realize a particular machine's potential, and a large body of work was produced to automatically derive parallel code from sequential program representations. It was during this time that David Kuck *et al.* at the University of Illinois produced the Parafrase compiler; Utpal Banerjee, also from the University of Illinois, formalized data dependence; and Ken Kennedy began work on the Parallel Fortran Compiler (PFC) at Rice.

By this time Larry Snyder, then at Purdue, already had significant experience in practical and theoretical aspects of parallel computation. It was his belief that automatic parallelization would not be the silver bullet many hoped it would be, and to date he is correct. In 1981, he began the BlueCHiP project to explore alternative architectures, programming methodologies and algorithms for exploiting parallelism. Snyder went to the University of Washington in 1983, and the project continued for another four years.

The BlueCHiP project was built around the CHiP reconfigurable parallel computer architec-

ture [Sny81]. A visual programming system called Poker was designed to exploit the CHiP architecture [Sny83, Sny84, SS86, NSS+88]. The system was built on a VAX11/780, which represented half of the Purdue Computer Science Department's computing power. The Poker environment exploited bit-mapped displays, which were still in their infancy. The Poker definition of the CHiP interconnection network was a precursor to the concept of the port ensemble in phase abstractions (described below).

Poker was designed exclusively for the CHiP architecture. It was for this shortcoming that Snyder began exploring machine abstraction. In 1986 he formalized the concept of a machine model, called a *type architecture*, and presented a candidate type architecture, or CTA, intended to abstract all modern parallel machines. The CTA and its unfortunate name have persisted, the former serving as the foundation of ZPL and then A–ZPL. The CTA was unique in that it did not provide the convenient shared memory view of parallel machines. A distributed memory view was taken, because it more accurately abstracted the great diversity of machines that existed [AS91, NS92].

Snyder believed automatic parallelization and a shared memory model were not the solution, so he and his colleagues developed the *phase abstractions* programming model [Sny90, AGNS90]. Phase abstractions provides a framework for structuring parallel programs, but it is not itself a programming language. Snyder and his students began experimenting with programming techniques and language abstractions founded on phase abstractions. David Socha's thesis developed Spot, the first in-depth foray into exploiting phase abstractions [Soc91]. Snyder, Ton Ngo, and Calvin Lin argued that a programmer may use a global view of the computation without a shared memory programming model [NS92], and Lin demonstrated this while maintaining portability across many different machines [LS91, Lin92, LS92].

In 1992, Snyder and Lin described Orca C, a variant of C built on phase abstractions [LS93]. The account was incomplete, and they next considered a subset of the language as a first step. This new language was intended for programming the third level of the phase abstractions' XYZ levels. For this reason the language was called ZPL, short for the Z-level programming language. A seminar in Spring 1993 was devoted to language design, and it was first presented the summer of the same year [LS94b]. Ruth Anderson, Bradford Chamberlain, Sung-Eun Choi, George Forman, Calvin Lin, Keith Partridge, Larry Snyder and W. Derrick Weathersby attended the seminar.

The original ZPL was primitive. Only a small class of data-parallel applications were well suited

for ZPL implementation, but those that were were expected to perform very well. The compiler was expected to be a simple source to source translator; sophisticated compilation techniques would not be necessary. Under the direction of Lin, the seminar attendees began the implementation in the summer of 1993, producing a prototype ready for evaluation the following summer [LS95]. The author joined the project at this time.

As the team became more aware of how to best compile for parallel machines, they added language features to improve the generality of the language, including flood regions (1996), multi-regions (1997), and support for pipelining wavefront computations (1999). At the same time, the compiler became more sophisticated, aggressively optimizing communication and array references.

In 1997, ZPL was a complete language for writing data-parallel applications, so it was released for public use [ZPL]. Since that time, the language has been further generalized and dubbed Advanced–ZPL, or A–ZPL. The design is still underway.

### *3.6 Summary*

A–ZPL arose from the needs of high performance parallel computing. We have looked at the role of machine, programming, and performance models in its design philosophy; and we have contrasted this to a number of other systems. We have also outlined the structure of the current University of Washington A–ZPL compiler and summarized the stages of the language's development.

# Chapter 4

# ARRAY STATEMENT FUSION AND ARRAY CONTRACTION

## *4.1 Motivation*

Array languages such as A–ZPL [Sny99], Fortran 90 (F90) [ABM+92], and High Performance Fortran (HPF) [Hig97] are important vehicles for expressing data parallelism. Though they simplify the specification of array-based calculations, they also present a potential problem: Large temporary arrays may need to be introduced, either by the programmer or by the compiler, because arrays are the basic unit of manipulation. *Array statement fusion* is a compiler transformation that groups array statements so that they are implemented by a single loop nest, in effect fusing the loop nests that implement each array statement. *Array contraction* is a transformation that converts array references within a loop nest to references of lower dimensional structures.

These temporary arrays frequently appear in the source program, but it is often convenient for the compiler to insert temporary arrays to preserve array language semantics or to simplify subsequent analyses (as described below). In both cases, these array temporaries increase memory use, degrade performance by polluting the data cache, and therefore make array languages impractical.

For example, the A–ZPL code fragment in Figure 4.1(a) uses temporary array `R` to cache a computation, while the C equivalent in Figure 4.1(b) uses only the scalar variable `s`, which can be viewed as a *contracted* form of the full array `R`. This array-to-scalar conversion, called *full contraction*, is the most common form of contraction. But there are also circumstances in which only a subset of an array's dimension may be contracted, called *partial contraction*. For example, consider the C code fragment in Figure 4.2(a).[1] The first dimension of array `A` represents an intermediate result. If only the final result is subsequently used (*i.e.*, the *n*th row in the sample code), the intermediate results need not be preserved, and the first dimension may be contracted, as in Figure 4.2(b).

In order address these deficits, an array language compiler must ensure that certain array lan-

[1] Chapter 5 shows how such codes may be expressed in A–ZPL.

```
[i,1..n] begin
           R := AA * D@north;
           D := 1.0 / (DD - AA@north * R);
           Rx := Rx - Rx@north * R;
           Ry := Ry - Ry@north * R;
         end;
```

(a)

```
for (j=1; j<=n; j++) {
  s = AA[i][j] * D[i-1][j];
  D[i][j] = 1 / (DD[i][j] - AA[i-1][j] * s);
  Rx[i][j] = Rx[i][j] - Rx[i-1][j] * s;
  Ry[i][j] =Ry[i][j] - Ry[i-1][j] * s;
}
```

(b)

Figure 4.1: Illustration of unnecessary array allocation (R) in an array language using a code fragment from the tridiagonal systems solver component of the SPEC FP95 Tomcatv benchmark.

```
for (j=1; j<=n; j++) {
  A[0][j] = ...;
}

for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    A[i][j] = foo(A[i-1][j]);
    ...
  }
}

for (j=1; j<=n; j++) {
  ... A[n][j] ...;
}
```

(a)

```
for (j=1; j<=n; j++) {
  A[j] = ...;
}

for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    A[j] = foo(A[j]);
    ...
  }
}

for (j=1; j<=n; j++) {
  ... A[j] ...;
```

(b)

Figure 4.2: Partial contraction code fragment sample.

guage statements are implemented with a single loop nest so that array references may be contracted. There are two approaches to this. The first is to scalarize the array language statement (*i.e.*, produce scalar loop nests for each array statement) and rely on a scalar language compiler to remove the array temporaries using its existing *scalar-level* optimizations. Specifically, the scalar compiler must fuse loops to enable contraction, as in Figure 4.1(b). The second approach is to optimize at the *array level* prior to scalarization (*i.e.*, perform analyses and transformations on array statements directly). Since fusion and contraction are mature and well understood transformations, the first approach would appear a natural choice because it simplifies the array language compilation process and leverages existing compiler technology.

However, we believe that the first approach is inferior for several reasons. First, optimizing array references by array contraction in a scalar compiler wastes compilation cycles, because scalar

languages do not necessitate the introduction of temporary arrays, so they infrequently appear. Second, removing array temporaries at the scalar level solves the problem at a greater conceptual distance from the source of the problem and at a greater cost. Third, though the issue of full contraction has been previously studied, partial contraction has not. Most importantly, it is impractical to implement an integrated optimization strategy when some optimizations (*e.g.*, communication pipelining) are performed at the array level, while others (*e.g.*, array contraction) are subsequently performed at the scalar level. Instead, we pursue array-level optimization in support of earlier claims that there are performance benefits to performing analyses and transformations at the array level [CCL+96, RK96].

In a comparative evaluation, we find that existing compilation systems generally fail to optimize these unnecessary temporary arrays and produce codes with correspondingly poor performance. We present techniques and the novel structures they employ, and we find that they consistently optimize the unnecessary arrays. We conclude that unnecessary arrays pose a significant, yet eminently tractable problem for array language compilers.

The next two sections define the representations used and give an abstract problem statement. Section 4.4 presents the approach to array contraction, and its evaluation appears in Section 4.5. The final two sections present related work and summarize.

## *4.2 Definitions*

This section describes the A–ZPL array statement normal form and array-level dependence representation. This representation is used throughout the compiler and to perform fusion and contraction. The construction and use of the dependence representation are also presented.

### *4.2.1 A–ZPL Array Statement Normal Form*

The *A–ZPL array statement normal form* permits array statements of two kinds: simple and complex. *Simple array statements* are element-wise functions of identically indexed arrays, all of the same rank. *Complex array statements* contain a single array operator (*e.g.*, `@`), the result of which is assigned to a scalar or array variable.[2] In both cases, a region, *REGION(s)*, is associated with each

[2]For simplicity, this discussion and Figure 4.3 do not address non-scalar left-hand-sides, such as record selection.

```
<statement>           ::= [<region>]   <simple-statement> ;
                        | [<region>]   <complex-statement> ;
<simple-statement>    ::= <id> := <simple-expression> ;
<simple-expression>   ::= <literal>  | <id>  | <unary-op> <simple-expression>
                        | <simple-expression> <binary-op> <simple-expression>
<complex-statement>   ::= <id> := <complex-expression> ;
<complex-expression>  ::= <id>@<id>
                        | +<<<id>  | max<<<id>  | ...          (reductions)
                        | +||<id>  | max||<id>  | ...          (scans)
                        | >>[<region>] <id>                     (flood)
                        | <id>#[<id-list>]                      (remap)
                        | ...
```

Figure 4.3: Informal summary of A–ZPL array statement normal form.

statement, *s*, to specify the extent of the computation. The grammar in Figure 4.3 describes A–ZPL array statement normal form, and sample A–ZPL code fragments before and after normalization appear in Figure 4.4.

A–ZPL statements are normalized via the introduction of temporary arrays. Region analysis is also necessary to determine the region governing the statement. In this work, we only need to determine when two regions are identical, which can usually be accurately determined via intra-procedural. The array statements of other languages (*e.g.*, HPF) may also be represented via the A–ZPL array statement normal form. In this case, more complex analysis is necessary, but it is comparable to the parallelization and data distribution tasks that the compiler must perform already.

This normal form is an appropriate representation for array statements because the data volume of each term in a single array statement is the same (*i.e.*, they are conformable). The normal form serves as an effective internal representation when compiling for parallel machines because it makes the alignment of arrays explicit. All array references are perfectly aligned and data transfer is only required for complex statements, so normalized statements compile to highly efficient parallel

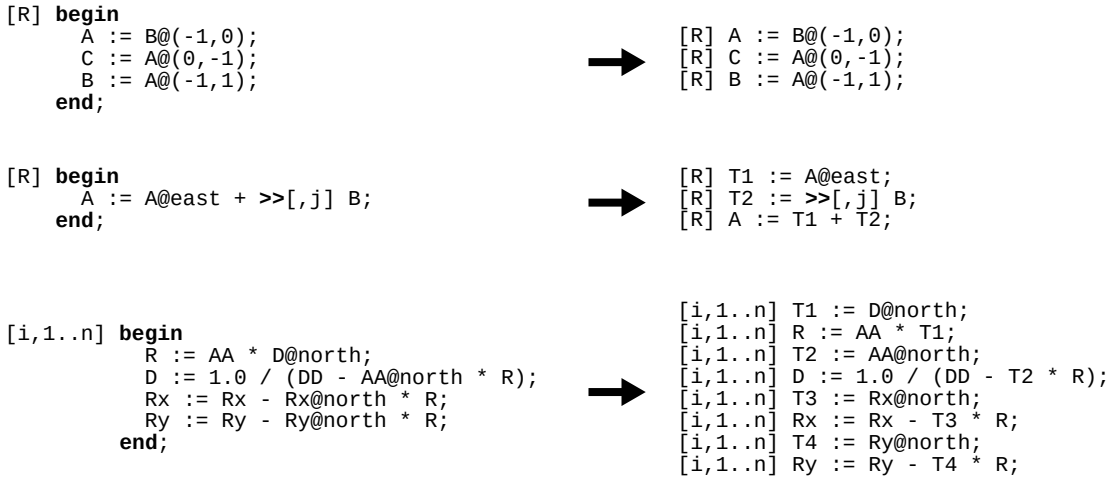



Figure 4.4: ZPL code fragments before and after normalization.

code [CCL+98b]. Normalized array statements contain only arrays of a single rank, which we designate the rank of the statement. Another implication of normalized array statements is that each simple statement can be implemented with a simple loop nest, and each complex statement can be implemented with a call to the A–ZPL runtime system.

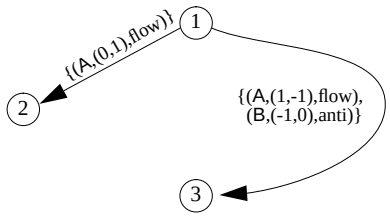
### *4.2.2 The Array Statement Dependence Graph*

In this section, we review the concept of data dependence, and we refashion existing mechanisms to represent dependences between normalized array statements. Data dependences [Wol96] represent ordering constraints on statements in a program. A *flow* or *true dependence* requires that a variable assignment precede a read to the same variable, and an *anti-dependence* requires the reverse. An *output dependence* requires that one assignment to a variable precede another assignment to the same variable, so that the appropriate value remains after both assignments. Transformations that reorder dependent statements (*i.e.*, move the dependence *target* before its *source*) are illegal, because they violate the dependence and do not preserve correctness.

Data dependence is also used to represent ordering constraints on iterations in the iteration space of a loop nest. The iteration space associated with a loop nest has a dimension for each loop in the nest. Loop transformations such as loop interchange or loop reversal are only legal if they

```
for (j=n; j>=1; j--) {
  for (i=n; i>=1; i--) {
    A[i][j] = B[i-1][j];
    B[i][j] = A[i-1][j+1];
  }
}
for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    C[i][j] = A[i][j-1];
  }
}
```

(a)




(b)

Figure 4.5: Two representations of the same (first) array computation from Figure 4.4: (a) C, and (b) array statement dependence graph.

preserve the data dependences in the iteration space. *Distance vectors* serve as a static analysis tool to represent data dependences concisely in an iteration space [Wol96].

**Definition 1** *A* distance vector *is an integer n-tuple,* $d = (d_1, d_2, \ldots, d_n)$*, representing a dependence between the iterations of a rank n iteration space, where the source of the dependence precedes the target by* $d_i$ *iterations in loop i (1 is the outermost), for* $1 \leq i \leq n$*. Note that a negative or zero value implies that the target precedes the source or that they are in the same iteration, respectively.*

The leftmost (*i.e.*, outermost) nonzero element of a distance vector determines which loop *carries* the dependence and by how many iterations. This element must be positive, otherwise the dependence target precedes the source in the loop that carries the dependence, which is clearly illegal. Thus, legal distance vectors are called *lexicographically nonnegative*.

Distance vectors are inappropriate for use in array-level compilation, because they are derived from loop nests, which are not created until after scalarization, the final step in array language compilation. As a result, we introduce a variant of the distance vector, called the *unconstrained distance vector*, to represent array-level data dependences between normalized array statements.

**Definition 2** *An* unconstrained distance vector *(UDV) is an integer n-tuple,* $u = (u_1, u_2, \ldots, u_n)$*, representing a dependence between two normalized n-dimensional array statements, where the source of the dependence precedes the target by* $u_i$ *iterations of the loop that iterates over dimension i (if both statements appear in the same loop nest).*

Dependences with nontrivial unconstrained distance vectors arise between statements containing arrays manipulated by the @ operator. UDVs are constructed by subtracting the dependence's target direction vector from its source direction. For example, the UDVs that arise from the dependences in the first code fragment of Figure 4.4 are $(0,0)-(0,-1)=(0,1)$ and $(0,0)-(-1,1)=(1,-1)$ for array A and $(-1,0)-(0,0)=(-1,0)$ for array B. The lexicographical nonnegativity of a UDV has no bearing on the legality of the dependence it represents.

Because scalarization of a normalized statement generates a single loop to iterate over the same dimension of all arrays in its body, we can characterize dependences by dimensions of the array rather than dimensions of the iteration space. Thus, $u_i$ is the distance of the dependence along array dimension $i$. UDVs are more abstract than traditional (constrained) distance vectors because they separate loop structure from dependence representation. Though UDVs are not fully general, they can represent any dependence that appears in our normal form.

We represent code using the *array statement dependence graph*.

**Definition 3** *An* array statement dependence graph *(ASDG), $G=(V,E)$, is a labeled, acyclic, directed graph, where a vertex v represents a set of statements, $s(v)$, edges represent data dependences between statements, and each edge, $(v_i,v_j)\in E$, is labeled, $l(v_i,v_j)$, with a set of (variable name, unconstrained distance vector, dependence type $\in$ {flow, anti, output}) tuples.*

An edge from $v_i$ to $v_j$, $(v_i,v_j)\in E$, in an ASDG indicates that there exists a dependence from a statement in $s(v_i)$ (the source of the dependence) to a statement in $s(v_j)$ (the target). The label on each edge describes the dependences the edge represents by naming the variables that induce the dependences and the associated UDVs and dependence types. Upon initial construction of an ASDG, each vertex $v_i$ represents exactly one statement; thus, $s(v_i)=\{s_i\}$. Only after statement fusion do vertices represent sets of greater than one member. Figure 4.5(b) contains the ASDG that corresponds to the first set of normalized array statements in 4.4.

### *4.2.3 Constructing an ASDG*

Next, we describe how an ASDG is constructed. First, define *USE(s)* to be the set of pairs $(x,R)$ such that the region $R$ indices of variable $x$ are *used* in statement $s$. Analogously, define *DEF(s)* to be the set of pairs $(x,R)$ such that the region $R$ indices of variable $x$ are *defined* in statement $s$. If

the indices of variable $x$ cannot be determined—either because the region associated with statement $s$ is not statically available or the remap operator is used with $x$—the region $R^*$ is used, denoting all indices. At a particular program point, a variable use is *upward exposed* if it can be reached from that point. An upward exposed use is represented with the tuple $(s,x,R)$, indicating that the use of the region $R$ indices of variable $x$ in statement $s$ is upward exposed. Let $UPUSE_{in}(s)$ and $UPUSE_{out}(s)$ be the set of upward exposed uses at the program point just before and after statement $s$, respectively.

We now describe the calculation of the $UPUSE_{in}(s)$ and $UPUSE_{out}(s)$ sets. First, define $SUCCESSORS(s)$ to be the set of all statements that may immediately follow statement $s$ in the dynamic execution of the program. Next, the following functions are used in a standard, backward, iterative, data-flow analysis.

$$UPUSE_{in}(s) = USE(s) \cup (UPUSE_{out}(s) - \{(s',x,R') | (s',x,R') \in UPUSE_{out}(s) \text{ and } (x,R) \in DEF(s) \text{ and } R' \subseteq R^3\}$$

$$UPUSE_{out}(s) = \bigcup_{s' \in SUCCESSORS(s)} UPUSE_{in}(s')$$

Analogously, the sets $REACHDEF_{in}(s)$ and $REACHDEF_{out}(s)$ are calculated via a forward, data-flow analysis. Again, regions are considered in the process of killing references.

$$REACHDEF_{in}(s) = \bigcup_{s' \in PREDECESSORS(s)} REACHDEF_{out}(s')$$

$$REACHDEF_{out}(s) = DEF(s) \cup (REACHDEF_{in}(s) - \{(s',x,R') | (s',x,R') \in REACHDEF_{in}(s) \text{ and } (x,R) \in DEF(s) \text{ and } R' \subseteq R\})$$

Now we construct ASDG $G = (V,E)$. If there are $n$ statements, let there be $n$ vertices, $s(v_i) = \{s_i\}$, and $l(v_i,v_j) = \emptyset$, for all $v_i, v_j \in V$. First, we construct the flow dependences. Edge $(v_i,v_j) \in E$ when there is a variable $x$ such that $(x,R) \in DEF(s_i)$, $(s_j,x,R') \in UPUSE_{out}(s_i)$, and $R \cap R' \neq \emptyset$. For each such edge, let $(x,u,\text{flow}) \in l(v_i,v_j)$, where UDV $u$ calculation is described above. We find a minimal solution. Analogously, the set of reaching definitions is calculated to define anti-

and output dependences. Specifically, $(x, u, \text{anti}) \in l(v_i, v_j)$ when $(v_i, v_j) \in E$, $(x, R) \in$ *USE($s_j$)*, $(s_i, x, R') \in$ *REACHDEF$_{in}$($s_j$)*, and $R \cap R' \neq \emptyset$; $(x, u, \text{output}) \in l(v_i, v_j)$ when $(v_i, v_j) \in E$, $(x, R) \in$ *DEF($s_j$)*, $(s_i, x, R') \in$ *REACHDEF$_{in}$($s_j$)*, and $R \cap R' \neq \emptyset$.

Dependence edges are necessarily conservative, for the region associated with a particular statement may not admit static analysis and we will use $R^*$. Nevertheless, the situation is far from bleak. A–ZPL regions are quite often static. Furthermore, region operators greatly simplify analysis. For example, regions `[R]` and `[east of R]` clearly do not intersect.

ASDGs are constructed for entire procedures, though statement fusion only operates on a single basic block at a time. In this context, basic blocks are defined at the source level (*i.e.*, before scalarization). Thus, a basic block is a single-entry, single-exit sequence of source-level instructions. An implication of this is that the control flow implicit in array statements does not break basic blocks.

As a notational convenience, *UPUSE$_{in}$($S = \{s_1, s_2, s_3\}$)* is defined to be *UPUSE$_{in}$($s_i$)*, where $s_i \in S$ lexically precedes all other statements in $S$. The same is true of other in-sets, and the dual is true of out-sets.

### *4.2.4 Using the ASDG*

If after scalarization, the source and target of a dependence appear in a single loop nest, a conventional (constrained) distance vector may be constructed from an unconstrained one given a description of the loop nest structure.

**Definition 4** *A* loop structure vector *is an integer n-tuple, $p = (p_1, p_2, \ldots, p_n)$, that describes the dimension and direction of each loop in an n-deep loop nest. Loop i (1 is the outermost loop in the loop nest) iterates over dimension $|p_i|$ in the direction of the sign of $p_i$, positive denoting increasing.*

A loop structure vector is a permutation of $(\pm 1, \pm 2, \ldots, \pm n)$. The loop structure vector that describes the loop nests in Figure 4.5(a) are $(-2, -1)$ and $(1, 2)$. In the first nest, the outer loop iterates over the second dimension and the inner loop iterates over the first dimension, both in a decreasing direction.

A constrained distance vector, $d = (d_1, d_2, \ldots, d_n)$, is constructed from an unconstrained one, $u$, and a loop structure vector, $p$, by letting $d_i = \frac{p_i}{|p_i|} u_{|p_i|}$, for $1 \leq i \leq n$. Consider the first and third statements in the first set of normalized array statements in Figure 4.4. If $p = (-2, -1)$, the UDVs $(-1, 0)$ and $(1, -1)$ become $(0, 1)$ and $(1, -1)$, respectively, when constrained. The constrained distance vectors are lexicographically nonnegative, so the dependences of the code in Figure 4.4 are preserved by the first loop nest in 4.5(a) resulting from loop structure vector $p$. There are no constraints on the structure of the second loop nest because it does not contain statements that depend on each other.

A *fusion partition* describes a particular fusing of the statements in an ASDG.[4]

**Definition 5** *A* fusion partition, $P = (P_1, P_2, \ldots, P_l)$, *of an ASDG,* $G = (V, E)$, *is a partitioning of the vertices of G into l disjoint sets,* $P_1, P_2, \ldots, P_l$, *called* fusible clusters *such that the following conditions hold:*

*(i) all statements in a single cluster operate under the same region (i.e., REGION($s_i$) = REGION($s_j$), $\forall s_i, s_j \in P_i$),*

*(ii) all unconstrained distance vectors on intra-fusible-cluster flow dependences are null vectors (i.e., $\forall P_i, \forall v_1, v_2 \in P_i$, if $(x, u, flow) \in l(v_1, v_2)$ then u is a null vector),*

*(iii) there are no inter-fusible-cluster cycles in G, and*

*(iv) a loop structure vector exists for each fusible cluster that preserves all intra-fusible-cluster dependences.*

Upon scalarization, all the statements in a fusible cluster are implemented with a single loop nest. The statements in each loop nest and the loop nests themselves are ordered by a topological sort using intra- and inter-fusible cluster dependence edges, respectively. The first condition above ensures that all the statements in a single cluster have the same (*i.e.*, conformable) loop bounds. The second condition ensures that no loop-carried flow dependences will inhibit parallelism. Inter-fusible-cluster dependences constrain the order of clusters, and the UDVs associated with inter-cluster dependences constrain the structure of the loop nest that implements them. Thus, the final two conditions ensure that inter- and intra-cluster dependences are preserved, respectively.

[4]This terminology is borrowed from Gao *et al.* [GOST92], who considered a similar problem. See Section 4.6.

An algorithm to decide the final condition is described in detail in Section 4.4.2. The *trivial fusion partition* of an ASDG is one in which there is exactly one statement in each fusible cluster.

Given a particular fusion partition we can decide for what arrays contraction has been enabled. First, we consider full contraction (*i.e.*, the case when an entire array becomes a scalar).

**Definition 6** *Given a fusion partition, $P = (P_1, P_2, \ldots, P_l)$, of an ASDG, $G = (V, E)$, references to array x in a fusible cluster $P_i$ is* contractible *if the following conditions hold:*

*(i) array x is dead at the entry and exit of $P_i$ (i.e., $\forall (v_1, v_2) \in E$, if $(x, u, flow) \in l(v_1, v_2)$, then $v_1 \in P_i \iff v_2 \in P_i$), and*

*(ii) the unconstrained distance vectors of all data dependences within cluster $P_i$ due to x are null vectors (i.e., $\forall v_1, v_2 \in P_i, (v_1, v_2) \in E$, if $(x, u, t) \in l(v_1, v_2)$, then u is a null vector).*

These conditions ensure that all dependent references to $x$ will appear in a single loop nest upon scalarization, and that there will be no loop-carried dependences due to $x$. The latter condition may be relaxed when the dependence is along a dimension of the array that is not distributed [CK94], but here we assume that all dimensions are distributed.

Next we determine when an array is a candidate for partial contraction. Because opportunities for partial contraction arise due to loop-carried dependences (see Figure 4.2), the second condition of Definition 5 must be excluded. Chapter 5 describes when this is necessary.

**Definition 7** *Given a fusion partition, $P = (P_1, P_2, \ldots, P_l)$, of an ASDG, $G = (V, E)$, dimension $j$ of array references to x in a fusion partition $P_i$ are contractible if the following conditions hold:*

*(i) there exists at least one intra-partition flow data dependence due to variable x and a single UDV, d, is associated with them all,*

*(ii) all entries of UDV d are zero, except $d_j$ (i.e., $d_j \neq 0$ and $d_i = 0, \forall i \neq j$),*

*(iii) for $R' = REGION(P_i)$[5] and $\forall (s, x, R) \in REACHDEF_{in}(P_i)$, $R \subseteq R'$, and*

*(iv) for $R' = REGION(P_i)$ and $\forall (s, x, R) \in UPUSE_{out}(P_i)$, $R \subseteq$* `[!`$d$ `in` $R'$`]`.

[5] This unambiguously identifies a single region, for all statements in a fusible cluster compute under the same region (see Definition 5).

The first condition ensures that there are both uses and definitions of array *x*, and that if references to it induce loop-carried flow data dependences, they are all the same. The second condition ensures that if a loop-carried flow dependence exists, it is cardinal (*i.e.*, only one non-zero element). The third condition ensures that only definitions of *x* that will be killed by the statements in $P_f$ reach them. And the final condition ensures that no values along the contracted dimension are required in later statements. Note that the elements of direction $!d$ are the negation of those of direction $d$.

### *4.3 Abstract Problem*

Statement fusion is performed both to enable the elimination of arrays by contraction and improve data cache utilization by exploiting inter-statement locality. For the first goal, we seek a fusion partition, *P*, for an ASDG, *G*, that minimizes the number of array references after array contraction (equivalently, maximize contracted references). A secondary, and complementary, goal is to minimize memory consumption. The number of array element references eliminated by the contraction of array *x*, called *reference weight*, $w(x, G)$, is a function of the number of times *x* is referenced at the array level and the size of the region under which these references occur. We call the sum of the reference weights of all contracted arrays the *contraction benefit* of a fusion partition. For the second goal, we seek a fusion partition that maximizes the number of arrays without inter-fusible-cluster dependences. The intuition is that while intra-cluster dependences are potential sources of cache reuse, we must be careful not to pollute the cache with the increased references that come with excessive fusion. When all references to an array appear in a single loop nest, all other loop nests are spared the cache burden of references to the array. Both problems are provably NP-complete, so we present approximate solutions in the next section.

### *4.4 Implementation*

This section presents algorithms for performing statement fusion to enable contraction and exploit locality. Certain fusion decisions may prohibit further fusion; thus, we chose to perform fusion for contraction before (and potentially at the expense of) fusion for locality. The former is likely to have greater impact on performance than the latter, and in practice they are not at odds. We also describe the details of scalarization.

### *4.4.1 Statement Fusion*

Before describing the algorithms, we introduce a definition. A *reference group* is a tuple $\langle x, S \rangle$ representing the fact that variable $x$ is referenced in the statements in set $S$. A *minimal* reference group contains a definition (statement) and all statements that use it and that it reaches. The references in reference groups serve as candidates for contraction; thus, the use of minimal reference groups improves the flexibility of contraction decisions by decreasing granularity.

Our algorithm to fuse statements to enable array contraction appears in Figure 4.6. It takes as input an ASDG, $G$, and it returns a fusion partition $P = (P_1, P_2, \ldots, P_l)$ containing $l$ clusters. $P$ is initialized to the trivial fusion partition, containing a cluster for each of the $l$ vertices in $V$ (line 2). The algorithm considers each reference group, $R_i$, which together represent all array references in $G$. The groups are considered in decreasing reference weight, $w(R_i)$, so that more frequently referenced variables are considered first. As a result, arrays that have potentially the largest single impact on the total contraction benefit are considered first. In line 7, set $c$ is assigned all the fusible clusters that contain references to variable $x$. The fusion of all the statements associated with the fusible clusters in $c$ might introduce inter-fusible-cluster cycles, so $c$ becomes the union of itself and the fusible clusters that are on inter-fusible-cluster cycles using the GROW function (line 8). This guarantees that there will be no dependence cycles, for cycles are not permitted in fusion partitions. If variable $x$ is contractible and a fusion partition is produced by combining all the vertices in $c$ (by Definitions 6/7 and 5), fusion is performed. The union of all statements in the clusters in $c$ is taken and assigned into the fusible cluster $P_k$. The counter $l$ is decremented to indicate that there are fewer clusters.

The FUSION-FOR-CONTRACTION algorithm uses three auxiliary routines. Function GROW$(c, G, P)$ returns the set of all fusible clusters in $P$ that are *(i)* not in $c$, *(ii)* reachable by a dependence path from a statement in $c$, and *(iii)* have a dependence path to a cluster in $c$. These are the statements that will be on an inter-fusible-cluster dependence cycle if the clusters in $c$ are fused. This function's running time is $O(e)$, where $e$ is the number of edges in $G$. The FUSION-PARTITION?$(c, G)$ and CONTRACTIBLE?$(x, c, G)$ predicates test the conditions in Definitions 5 and 6/7, respectively. They both run in $O(e)$ time. The former function can ignore inter-cluster cycles because line 8 guarantees they will not exist. It also calls FIND-LOOP-STRUCTURE (de-

INPUT: $G = (V, E)$ : an array statement dependence graph
OUTPUT: $P = (P_1, P_2, \ldots, P_l)$ : a fusion partition of $G$

FUSION-FOR-CONTRACTION($G$)
1 $l \leftarrow |V|$
2 $P \leftarrow$ trivial partition of $G$ $\quad \{P_i = \{S_i\}, 1 \leq i \leq l\}$
3 $R \leftarrow$ reference groups in $G$ $\quad \{$list of tuples $\langle x, S\rangle$ indicating array $x$ appears in statements $S\}$
4 $R \leftarrow$ sort $R$ by decreasing weight $w$ $\quad \{w(R_i) \leq w(R_j)$ for $i \leq j\}$
5 **for** $i \leftarrow 1$ **to** $|x|$ **do** $\quad \{$consider variable in reference group $i$ for contraction$\}$
6 $\quad \langle x, S\rangle \leftarrow R_i$ $\quad \{$consider variable $x$, which appears in the statements of set $S\}$
7 $\quad c \leftarrow \{P_j | \exists s \in S$ such that $s \in s(v_j)\}$
8 $\quad c \leftarrow c \cup$ GROW$(c, G, P)$
9 $\quad$ **if** CONTRACTIBLE?$(x, c, G)$ and FUSION-PARTITION?$(c, G)$ **then**
10 $\quad\quad k \leftarrow$ smallest $j$ for $P_j \in c$
11 $\quad\quad s(v_k) \leftarrow \cup_{z \in c} s(z)$
12 $\quad\quad l \leftarrow l - (|c| - 1)$
13 $\quad$ **end if**
14 **end for**
15 **return** $P$

Figure 4.6: Algorithm to find a fusion partition that enables contraction in an ASDG.

scribed in the next section) to decide whether condition *(iv)* of Definition 5 is met. If there are $n$ reference groups in $G$, the total running time for FUSION-FOR-CONTRACTION is $O(ne)$.

The algorithm to perform fusion for locality enhancement is identical to that in Figure 4.6, except that the CONTRACTIBLE? predicate in line 7 is eliminated. We try to fuse all statements that reference the array that will have the greatest single locality benefit, which is analogous to the contraction benefit. Next, we will describe the process by which an ASDG is scalarized given a fusion partition.

### *4.4.2 Scalarization*

Scalarization generates a loop nest for each fusible cluster in a fusion partition. The generated loop nest and statements in the loop nests are ordered by a topological sort using inter- and intra-fusible-cluster dependences, respectively. The only work remaining is deciding the structure of each loop

nest, *i.e.*, the direction in which and dimension over which each loop iterates. This information is encoded in a loop structure vector (Definition 4) for each fusible cluster. Intra-cluster dependences constrain the structure of the loop nest that will implement its statements (*i.e.*, the loop nest must preserve these dependences). When the dependences do not fully constrain the structure of the loop nest, we will favor the loop structure that best exploits spatial locality.

The algorithm to find a loop structure vector given a set of unconstrained distance vectors from intra-fusible-cluster array-level dependences appears in Figure 4.7. FIND-LOOP-STRUCTURE consists of a doubly nested loop. The outer loop (line 4) iterates over the loops of the target loop nest, and the inner loop iterates over the dimensions of the arrays. The loop body matches loops to array dimensions (lines 9 through 13). We consider target loops from outer to inner because when a dimension is assigned to a loop, the dependences that are carried in that loop do not constrain the structure of the inner loops (thus, set $C$ is pruned in line 12). We consider dimensions from 1 to $n$ so that inner loops will be matched with higher array dimensions to exploit spatial locality (assuming row-major allocation), if allowed by the constraints. For example, suppose set $C$ contains two UDVs $(1,0)$ and $(-1,1)$. The outer loop can not iterate over the first dimension, because the 1 and $-1$ imply opposite iteration directions. But the outer loop can iterate over the second dimension, eliminating the second UDV (line 12) and allowing the the inner loop to iterate over the first dimension. The returned loop structure vector is $p = (2,1)$.

If there are $e$ dependences, the running time of lines 8 and 12 is $O(e)$, so FIND-LOOP-STRUCTURE runs in $O(n^2e)$ time. Because the rank of the arrays, $n$, is typically very small and effectively constant [SLY90], the algorithm is essentially linear, $O(e)$, in the number of dependences.

### *4.5 Performance*

This section evaluates the preceding approach to statement fusion and array contraction—as implemented in the A–ZPL compiler—by comparison to commercial F90/HPF compilers and hand-coded C code. Furthermore, we examine the transformations' effect on memory use and their relative impact on run-time performance. Finally, we evaluate how their interaction with communication optimizations effect performance.

INPUT: $C$ : a set of $m$ unconstrained distance vectors, each of size $n$
OUTPUT: $p$ : a loop structure vector of size $n$ (loop $i$ iterates over array dimension $|p_i|$ in the direction of the sign of $p_i$)

FIND-LOOP-STRUCTURE($C$)

```
1  for j ← 1 to n do      {initialize unassigned mask}
2     b_j ← true      {b_j = true ⇒ array dimension j has not yet been assigned to a loop}
3  end for
4  for i ← 1 to n do      {iterate over loops}
5     solution ← false;
6     for j ← 1 to n do      {iterate over array dimensions}
7        if b_j then
                          ⎧ +1   if ∀u ∈ C, u_j ≥ 0
8           d ← ⎨ −1   if ∀u ∈ C, u_j ≤ 0 and ∃u ∈ C, u_j < 0
                          ⎩  0   otherwise
9           if d ≠ 0 then      {can loop i iterate over dimension j?}
10             b_j ← false
11             p_i ← jd
12             C ← C − {u ∈ C | u_j ≠ 0}
13             solution ← true
14             continue i loop
15          end if
16       end if
17    end for
18    if !solution then
19       return NOSOLUTION      {no dimension found for loop i}
20    end if
21 end for
22 return p
```

Figure 4.7: Algorithm to find a legal loop structure vector given a set of unconstrained distance vectors from intra-fusible-cluster data dependences.

The benchmark programs we use to evaluate our transformations represent typical parallel array language programs. The SP application and EP kernel belong to the NAS parallel benchmark suite [BBB+94, BHS+95]. SP solves sets of uncoupled scalar pentadiagonal systems of equations; it is representative of portions of CFD codes. EP generates pairs of Gaussian random deviates, and it is considered "embarrassingly parallel." EP characterizes the peak realizable FLOPS of a parallel machine. Tomcatv is a SPEC FP95 benchmark that performs vectorized mesh generation. The Simple code solves hydrodynamics and heat conduction equations by finite difference methods [CHL78]. The Fibro application uses mathematical models of biological patterns to simulate the dynamic structure of fibroblasts [DLMW95]. The Frac code generate fractal images. These benchmarks only benefit from full contraction (*i.e.*, all array dimensions). Chapter 6 gives partial contraction performance data.

### *4.5.1 Comparison to Commercial Compilers*

In order to assess the state of the art, we determine how aggressively current commercial array language compilers perform statement fusion and array contraction. We examine compilers for F90 and HPF (a parallel superset of F90) because F90 is the array language to which the greatest development effort has been devoted.

The developers of commercial compilers do not advertise the specific optimizations that their products perform, so we infer their ability to perform statement fusion and array contraction by studying compiler output for a set of carefully selected code fragments, shown in Figure 4.8. In all cases, arrays `B`, `T1` and `T2` are not live beyond the given code fragments, while `D` is. The fragments in (1), (2) and (3) test a compiler's ability to perform statement fusion to exploit temporal locality. The fragments differ in the data dependences they contain. The fragments in (4) and (5) test a compiler's ability to eliminate compiler temporaries, and (6) and (7) test the same for user temporaries, in this case array `B`. Fragment (8) contains two user arrays that can be contracted if contraction of the compiler array for the third statement is sacrificed. The fragment tests whether a compiler properly weighs this tradeoff. Figure 4.1 summarizes whether each compiler properly fused (and in some cases contracted arrays) in each code fragment.

First, observe that the PGI and IBM compilers appear not to perform any statement fusion (*i.e.*,

```
D(1:n,1:m) = A(1:n,1:m)+A(1:n,1:m)
C(1:n,1:m) = A(1:n,1:m)*A(1:n,1:m)                                   (1)

D(1:n,1:m) = A(0:n-1,1:m)+A(0:n-1,1:m)
C(1:n,1:m) = A(1:n,1:m)*A(1:n,1:m)                                   (2)

D(1:n,1:m) = A(0:n-1,1:m)+C(0:n-1,1:m)
C(1:n,1:m) = A(1:n,1:m)*A(1:n,1:m)                                   (3)

A(1:n,1:m) = A(1:n,1:m)+A(1:n,1:m)                                   (4)

A(1:n,1:m) = A(0:n-1,1:m)+A(0:n-1,1:m)                               (5)

B(1:n,1:m) = A(1:n,1:m)+A(1:n,1:m)
C(1:n,1:m) = B(1:n,1:m)                                              (6)

B(1:n,1:m) = A(1:n,1:m)+A(1:n,1:m)+C(0:n-1,1:m)
C(1:n,1:m) = B(1:n,1:m)                                              (7)

T1(1:n,1:m) = B(1:n,1:m)
T2(1:n,1:m) = B(1:n,1:m)
A(1:n,1:m) = A(2:n+1,1:m) + T1(2:n+1,1:m) + T2(2:n+1,1:m)            (8)
```

Figure 4.8: Code fragments to evaluate Fortran 90 and HPF compiler optimizations.

each array statement compiles to a single loop nest). The implementors hoped to leverage the optimizations performed by the back end Fortran 77 compiler, which does in fact perform fusion. Unfortunately, the back end compiler does not perform contraction because it was not designed to compile scalarized array language programs. Most of the compilers successfully eliminate compiler temporaries. This is not surprising given that it requires only a simple local analysis, but additional experiments (Section 4.5.4) show that this transformation alone is not sufficient. Though the APR compiler appears to perform fusion for locality and compiler array contraction, it is unable to fuse loops that carry anti-dependences.

Finally, notice that the Cray F90 compiler appears to perform both statement fusion and array contraction, but there are circumstances under which it fails. The compiler is unable to fuse statements where the resulting loop nest would contain loop-carried anti-dependences. As a result, fusion does not occur in either (3) or (7), in the latter case inhibiting contraction. We also infer that the compiler considers contraction of compiler and user temporary arrays separately, since it

Table 4.1: Observed behavior of five array language compilers. A $\surd$ indicates that the compiler produced the proper fused/contracted code.

| | *fusion* | | | *compiler temps* | | *user temps* | | *trade-off* |
|---|---|---|---|---|---|---|---|---|
| *compiler* | *(1)* | *(2)* | *(3)* | *(4)* | *(5)* | *(6)* | *(7)* | *(8)* |
| PGI HPF 2.1 | | | | $\surd$ | $\surd$ | | | |
| IBM XLHPF 1.2 | | | | $\surd$ | $\surd$ | | | |
| APR XHPF 2.0 | $\surd$ | $\surd$ | | $\surd$ | | | | |
| Cray F90 2.0.1.0 | $\surd$ | $\surd$ | | $\surd$ | $\surd$ | $\surd$ | | |
| A–ZPL 1.13 | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ |

contracts the compiler temporary in (8) at the expense of contracting the two user temporaries. The Cray compiler probably never inserts compiler temporaries when a single statement does not require it, even if this transformation would enable the contraction of multiple other arrays. The technique we describe always inserts compiler arrays, and it treats compiler and user arrays together as candidates for contraction. If a single statement does not truly require a compiler array, our algorithm is guaranteed to contract it unless a more favorable contraction is performed that prevents it.

### *4.5.2 Comparison to Hand-coded*

A successful array language compiler will produce scalar code comparable to that of a skilled scalar language programmer. We now compare code produced by the A–ZPL compiler with equivalent programs written in a scalar language. Figure 4.2 summarizes for each of the six benchmarks the number of static arrays appearing in the compiled code with and without array contraction. Note that within each code, nearly all arrays are approximately the same size. We see that all compiler-generated arrays have been eliminated. The benefit of this is that programmers can better comprehend the memory use of their code when the compiler only infrequently introduces arrays. Figure 4.2 shows a substantial reduction in the number of static arrays. All the arrays are eliminated in EP, and in all but one of the other benchmarks more than half are eliminated.

The final column in Figure 4.2 gives the number of arrays that appear in equivalent scalar language codes. The scalar language codes are all publicly available C or Fortran 77 programs written by third parties. The compiler-generated code has the same or fewer arrays on all the benchmarks

Table 4.2: Static arrays contracted (categorized as compiler/user arrays). Fibro was developed in A–ZPL, so no equivalent scalar version exists.

| | *array language* | | | *scalar* |
|---|---|---|---|---|
| *application* | *w/o contr.* | *w/ contr.* | *% change* | *lang.* |
| EP | 22(0/22) | 0(0/0) | -100.0 | 1 |
| Frac | 8(0/8) | 1(0/1) | -87.5 | 1 |
| SP | 181(18/163) | 56(0/56) | -69.1 | 48 |
| Tomcatv | 19(4/15) | 7(0/7) | -63.2 | 7 |
| Simple | 85(20/65) | 32(0/32) | -62.4 | 32 |
| Fibro | 49(0/49) | 27(0/27) | -44.9 | n/a |

except SP, which requires a form of partial contraction not yet supported. Despite this shortcoming, SP still benefits from a substantial performance improvement, as we see in Section 4.5.4.

### *4.5.3 Effect on Memory Usage and Problem Size*

While the preceding section uses static array counts to suggest that contraction conserves memory, here we consider dynamic data to discover more precisely how memory conservation from array contraction enables larger problems to be solved in a fixed amount of memory. The degree by which contraction allows larger problems to be solved is an important issue for memory bound applications. In the following we assume that for a single program on a particular machine: *(i)* all arrays are the same size, which we call the *problem size*, *(ii)* all array elements are the same size, and *(iii)* a constant amount of memory is available for array allocation independent of problem size. The degree by which the maximum problem size scales due to contraction is the ratio of the maximum problem size after and before contraction, $\frac{s_a}{s_b}$. Given the above assumptions and that maximum problem size is inversely proportional to the maximum number of simultaneously live arrays, $l$, the scaling factors becomes $\frac{l_b}{l_a}$. We subtract 1 and multiply by 100 to convert the maximum problem size scaling factor to percent change,

$$C(l_b, l_a) = 100 \times \frac{l_b - l_a}{l_a}.$$

The first columns of Figure 4.3 give the dynamic $l_b$ and $l_a$ values and the calculated $C$ value for each benchmark.

Table 4.3: Effect of contraction on maximum achievable problem size on single IBM SP-2 and Cray T3E nodes.

| | | | | IBM SP-2 maximum problem size | | | Cray T3E maximum problem size | | |
|---|---|---|---|---|---|---|---|---|---|
| *app.* | $l_b$ | $l_a$ | $C$ | *w/o contr.* | *w/ contr.* | *% change (vol)* | *w/o contr.* | *w/ contr.* | *% change (vol)* |
| EP | 22 | 0 | $\infty$ | $2^{19}$ | $\infty$ | $\infty(\infty)$ | $2^{16}$ | $\infty$ | $\infty(\infty)$ |
| Frac | 8 | 1 | 700.0 | $1531^2$ | $5730^2$ | 274.3(1300.7) | $1409^2$ | $3987^2$ | 183.0(700.7) |
| Tomcatv | 19 | 7 | 171.4 | $929^2$ | $1530^2$ | 64.7(171.2) | $1293^2$ | $2128^2$ | 64.6(170.9) |
| Fibro | 49 | 27 | 81.5 | $583^2$ | $790^2$ | 35.5(83.6) | $572^2$ | $774^2$ | 35.3(83.1) |
| SP | 23 | 17 | 35.3 | $74^3$ | $81^3$ | 9.5(31.1) | $91^3$ | $101^3$ | 11.0(37.7) |
| Simple | 40 | 32 | 25.0 | $640^2$ | $715^2$ | 11.7(24.8) | $623^2$ | $702^2$ | 12.7(27.0) |

To confirm the above analysis, we experimentally determine for each benchmark the largest problem size that fits on a single node of the Cray T3E and the IBM SP-2. Both machines have operating system facilities to limit the process size, so we found the largest problem size that does not result in a memory allocation failure. Columns seven and ten of Figure 4.3 give the change in problem size, both along one dimension of the problem domain and in total data volume. The experimental data shows that these applications respect the above assumptions, for the $C$ value accurately predicts the change in problem volume. The one exception is Frac on the SP-2, which violates assumption *(ii)*. EP, in which all arrays are eliminated, clearly benefits the most from contraction because the contracted form uses a constant amount of memory, independent of the problem size. The other applications' changes in problem size vary from 10% to 274% along a single dimension or 25% to 1300% in total volume.

### *4.5.4 Run-time Performance*

This section considers the run-time performance impact of array contraction and statement fusion.[6] Though we discuss only the relative effect of these transformations, other studies have shown that the A–ZPL compiler produces code that performs within 10% of hand-coded C plus message passing and generally better than HPF [CCL+98a, LSA+94, LS94a, Ngo97].

In order to better understand the performance contributions of fusion and contraction, we measure execution time using several different optimization strategies.

[6] Additional performance data appears in an earlier article by the author [LLS98].

**baseline** : no fusion or contraction transformations are performed

**cc** : only compiler inserted arrays are considered for array contraction

**ca** : all arrays are considered for array contraction

**ca+f** : all arrays are considered for contraction and additional statement fusion is performed to improve locality

Figures 4.9, 4.10 and 4.11 show the execution time improvement of each transformation relative to the *baseline* for each benchmark for a varying number of processors on the Cray T3E, IBM SP-2 and Intel Paragon. Execution times are the best of three trials on the T3E and Paragon and of at least six trials on the SP-2, a machine that suffers from great performance variance from trial to trial. So that we may neutralize the effect of communication masking all other performance characteristics on large processor sets, we scale the problem sizes with the number of processors (*i.e.*, the amount of data per processor remains constant as the number of processors increases).

These graphs demonstrate that performing contraction on both compiler and user arrays in array languages is essential. The predominant characteristic of the graphs is that *ca* dominates the other transformations. The elimination of a large portion of the compiler and user arrays by contraction drastically improves temporal locality, always resulting in a significant performance boost (up to 400% on one application). Fibro on the SP-2 does not benefit from contraction for large number of processors because of interactions with communications optimizations, discussed in the next section. In the larger applications, contraction of only compiler arrays, *cc*, provides a substantive performance enhancement (up to 30%), but it is only a fraction of the potential contraction benefit. The smaller benchmarks, such as Fibro, EP and Frac, require no compiler arrays, so they do not benefit from *cc*. Clearly, transformation *cc* does not sufficiently address the problem of unnecessary temporary arrays in array languages.

As the number of processors, $p$, varies, certain trends become evident. The improvement due to contraction in EP and Frac is effectively independent of the number of processors because these codes scale nearly perfectly with $p$. The improvement due to fusion and contraction grows with $p$ for some programs, such as Simple and Tomcatv on the SP-2, when the transformations improve

portions of the program that make up a larger fraction of total execution time as $p$ grows (*i.e.*, the transformations improve portions of the code that do not scale well with $p$).

The performance improvement for a transformation decreases with $p$ when the transformation improves a portion of the code that makes up a smaller fraction of total execution time as $p$ increase. This happens when some other segment of the code is not scaling well and consumes a larger fraction of total execution time as $p$ increases. SP exhibits this behavior because only portions of the code that scale well benefit from the transformations. When both scaling and non-scaling segments of a code benefit from the transformations, machines characteristics (*e.g.*, the relative costs of cache misses, communication and floating point operations) dictate the trends. This is exemplified by Tomcatv, which shows level, increasing and decreasing trends on the three machines in our experiments.

In these experiments, $ca{+}f$ produces very little benefit beyond that of $ca$. This is because the great many opportunities for contraction result in very aggressive fusion.

### *4.5.5 Interaction with Communication Optimization*

In this section, we demonstrate that statement fusion interacts with communication optimizations and for this reason should be performed at the array level. Some optimizations cannot be performed practically at the scalar level because they interact with other transformations that can only occur at the array level. If an optimization that interacts with array-level transformations is relegated to a scalar compiler, either the array-level transformations must understand and reason about the optimization behavior of the scalar compiler or vice versa. It is unlikely that scalar compilers can understand the optimization strategy of all the compilers that compile to it, so the array compiler must consider scalar optimizations when performing array transformations, effectively moving the scalar transformations into the array compiler.

To achieve efficient parallel execution, compilers must often perform aggressive communication optimizations [CS97], such as redundancy elimination, message combining and pipelining. In some cases, these communication optimizations are at odds with fusion for contraction. For example, pipelining hides latency by separating the send and receive portions of communication with computation, but fusion may collect into a single loop some of the statements that could be used to hide

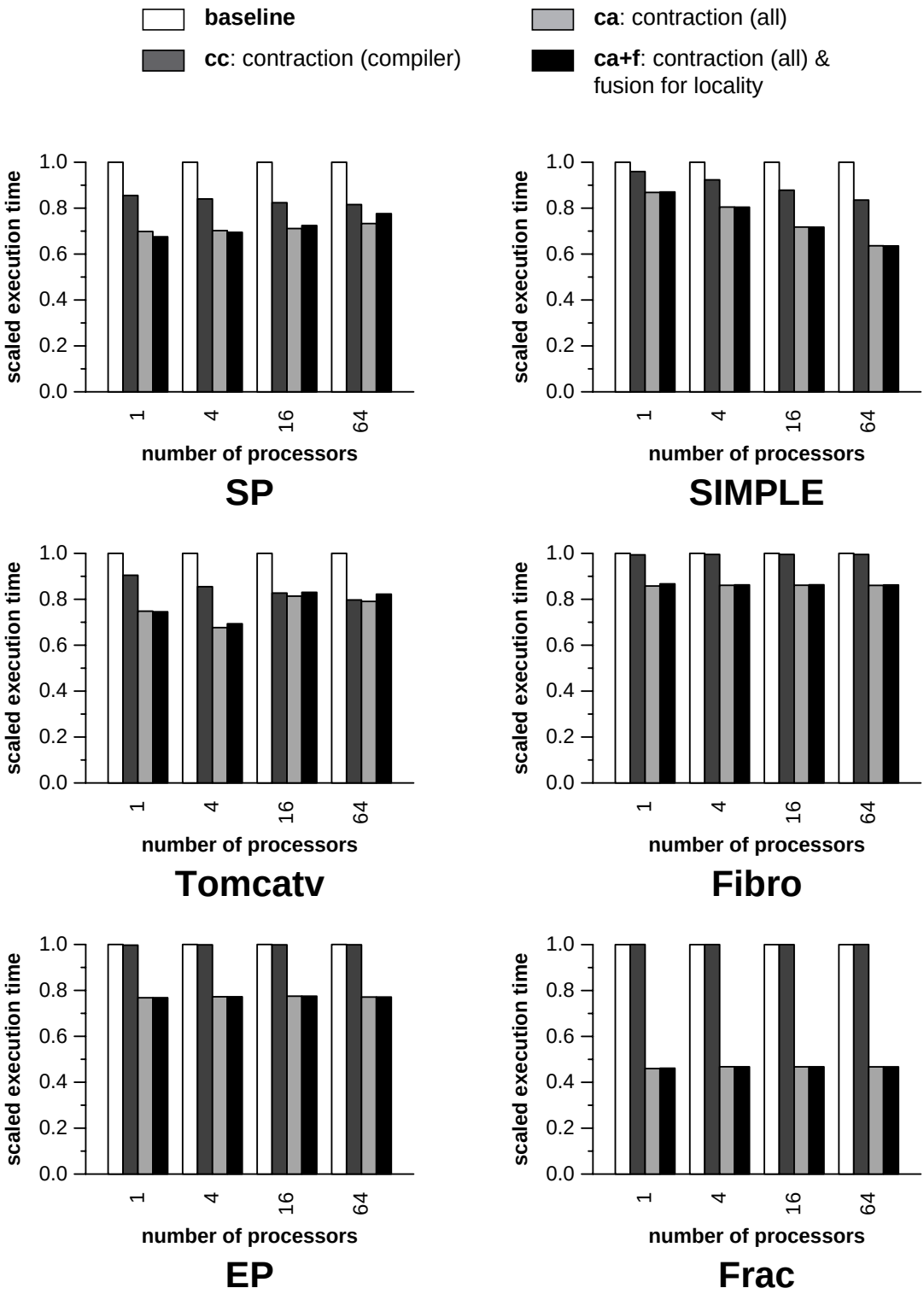



Figure 4.9: Benchmark performance on Cray T3E.

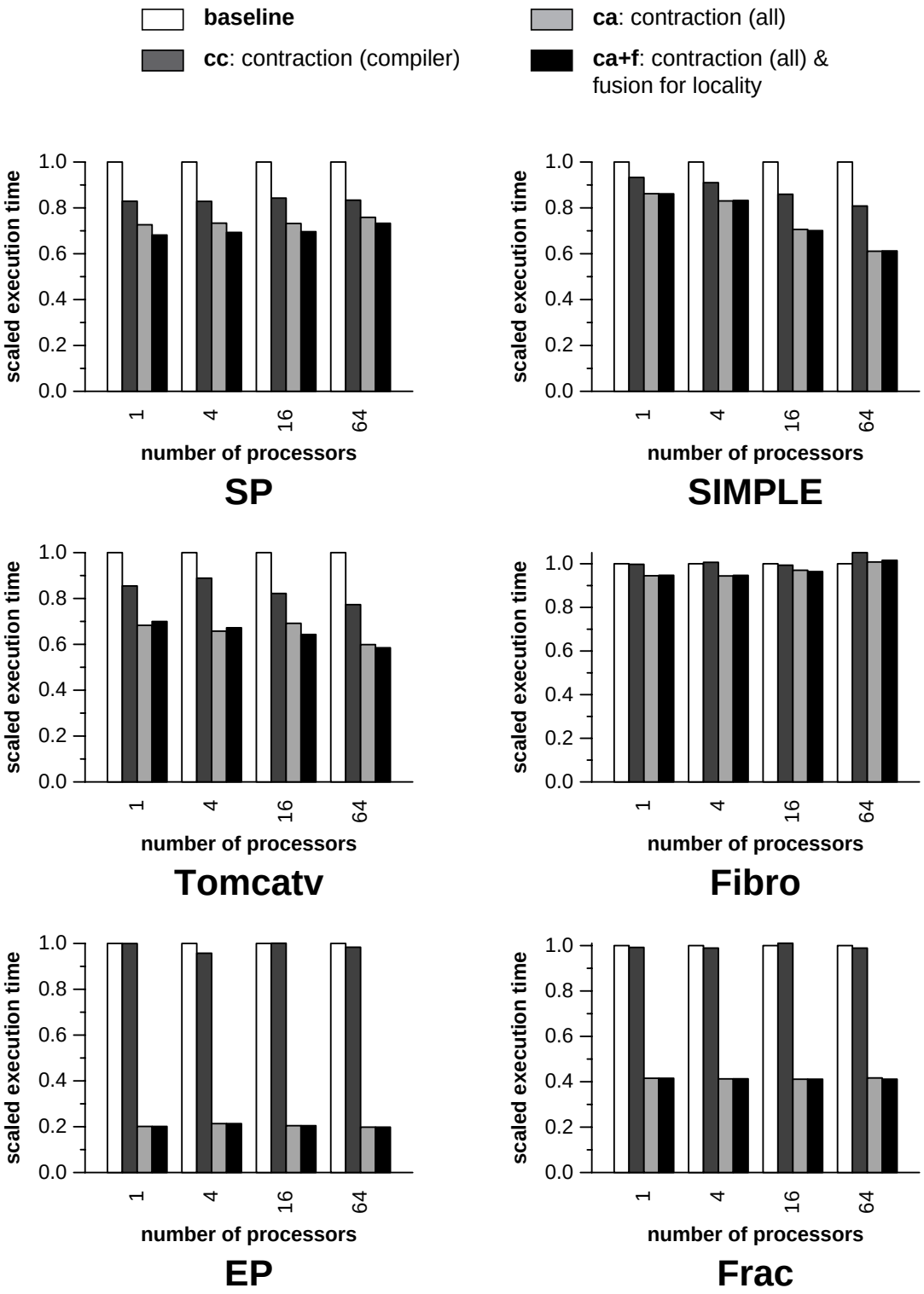



Figure 4.10: Benchmark performance on IBM SP-2.

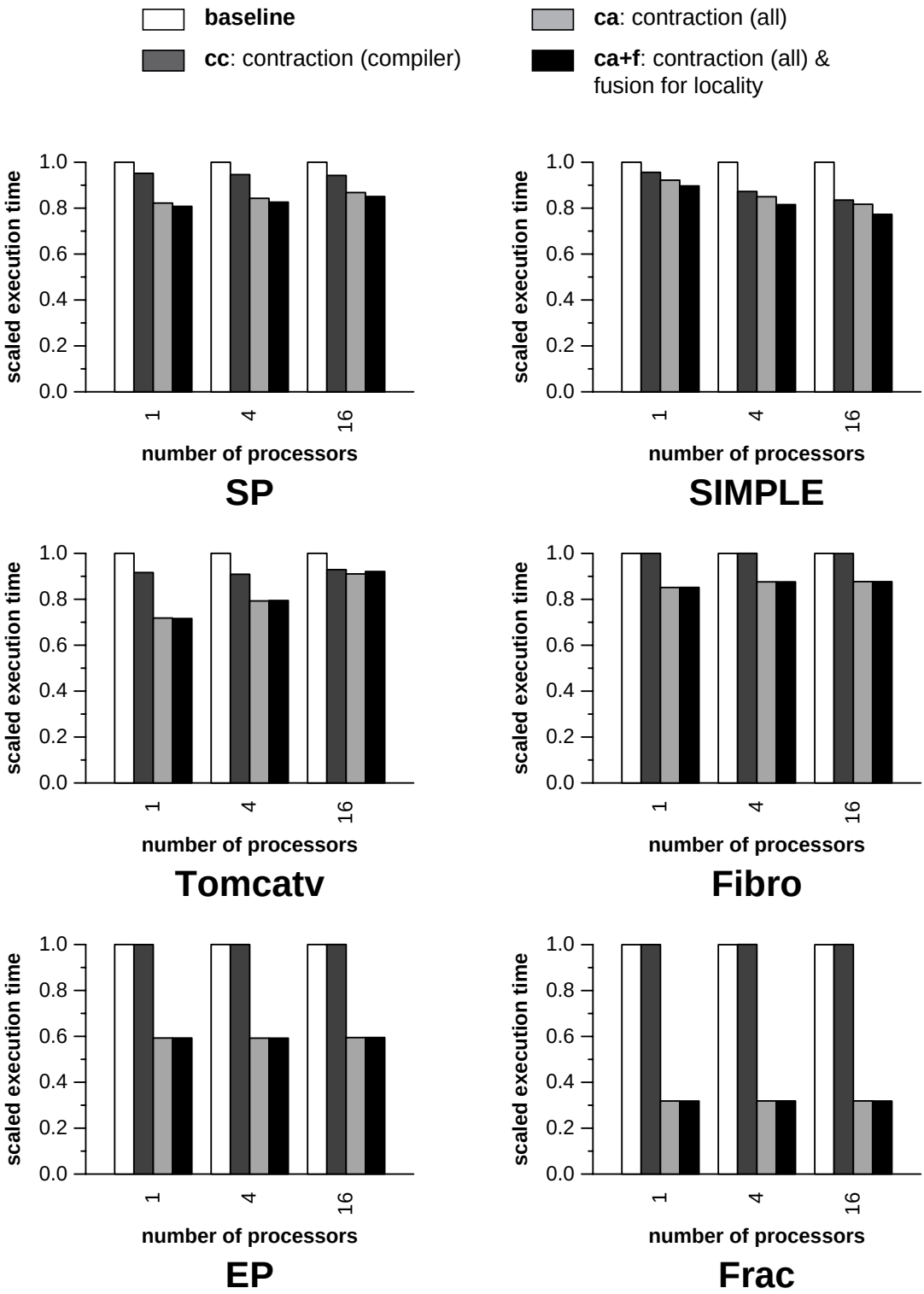



Figure 4.11: Benchmark performance on Intel Paragon.

latency, potentially disabling overlap. The experiments presented thus far resolve this conflict by favoring fusion, *i.e.*, fusion is never prevented by communication optimizations. We consider an alternative strategy in which communication optimizations are favored, *i.e.*, fusion cannot be performed if it reduces the benefit of communication optimization. This simulates the performance of a compiler that performs contraction *after* scalarization. Note that message vectorization never conflicts with fusion, so it is always performed.

As the amount of fusion increases, the potential for conflict with communication optimization grows. Figure 4.12 illustrates this effect. On the T3E, when favoring communication optimizations over fusion for contraction, Simple, Tomcatv, SP and Fibro suffer a slowdown of 25.4%, 22.7%, 9.6% and 5.1%, respectively. On the SP-2, they slowdown by 31.8%, 66.5%, 10.5% and -10.6%, respectively. On the Paragon, they slowdown by 7.5%, 8.5%, 5.0% and 0.9%. The first three programs slowdown significantly because the communication optimizations disable a large number of array contraction opportunities without producing comparable communication benefits. Only one fusion for locality opportunity and no contraction opportunities are lost by favoring communication optimizations in Fibro. It slows down little and in one case it speeds up, because of the additional communication optimization. EP and Frac do not slowdown because they are small codes that do not benefit from communication optimization, with or without fusion.

We have not demonstrated that favoring contraction is optimal, but we have shown that if a choice is to be made, fusion for contraction should be favored. This suggests that it would be very difficult to perform communication optimizations if fusion and contraction occur after scalarization. The communication transformations would have to understand contraction well enough to optimize without disabling it, since it is unlikely that the scalar compiler could reason about communication primitives once they are scalarized. The Fibro data suggests that there are delicate tradeoffs that only an integrated approach to fusion and communication optimization can address, which would further complicate performing fusion at the scalar level. Furthermore, we expect to find that integration will become even more important on machines with low cost synchronization in hardware (*e.g.*, SGI Origin, Sun E10000). Thus, these results support our claim that these optimizations for array languages should be performed at the array level, for this is the only practical point during which they may be performed together.

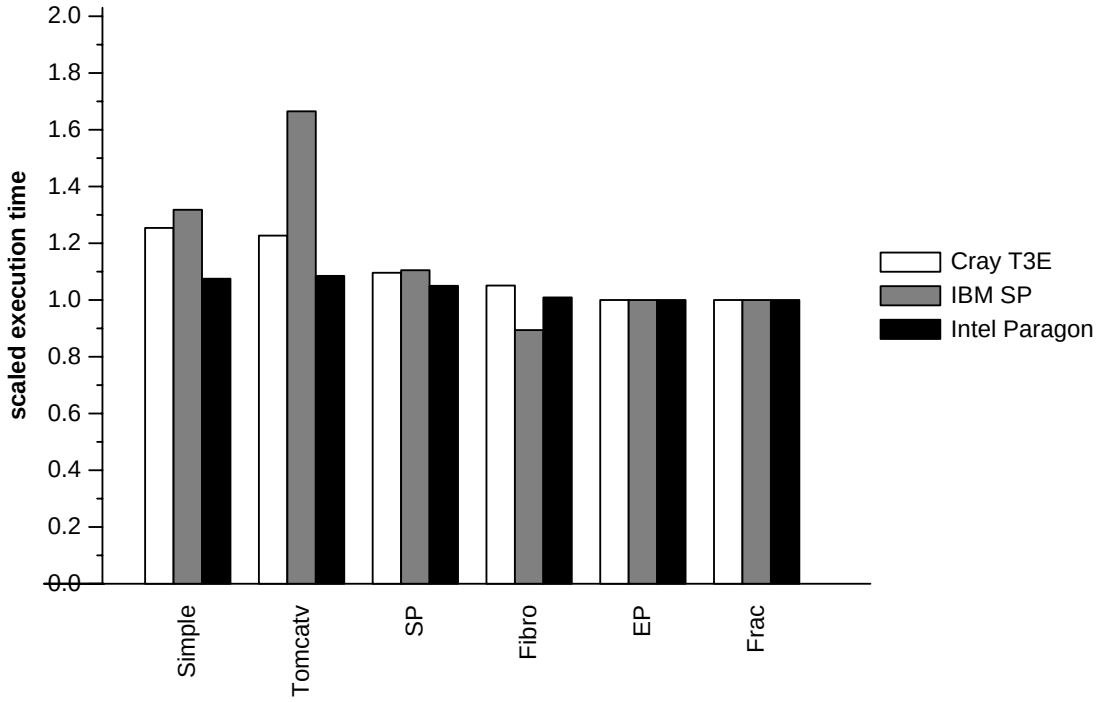



Figure 4.12: Execution time when communication optimization is favored.

### *4.6 Related Work*

The problem of optimizing array languages at the array level has recently received attention by others. Hwang *et al.* describe a scheme for *array operation synthesis* [HLJ95]. Multiple instances of element-wise F90 array operations such as MERGE, CSHIFT, and TRANSPOSE are combined into a single operation, reducing data movement and intermediate storage. Their work does not address the inter-statement intermediate array problem except to substitute an intermediate array's use by its definition. This *statement merge* optimization [Ju92] enables more operation synthesis, but it is not always possible, and it potentially introduces redundant computation and increases overall program execution time. Roth and Kennedy have independently developed a similar array based data dependence representation for F90, and they describe its use in scalarization [RK96]. They do not address the fusion for contraction problem.

Loop fusion in the context of scalar programming languages such as Fortran 77 is well un-

derstood [Wol96]. Though most work only considers pairwise fusion, some research addresses collective loop fusion, as we do. Sarkar and Gao [SG91] transform loop nests by loop reversal, interchange and fusion to enable array contraction. They target multiprocessors and exploit pipelining by executing producer and consumer loops on different processors, so they are free to ignore all but flow dependences. Because we choose to distribute iteration spaces instead, preservation of all types of dependences is critical to our solution. Gao *et al.* [GOST92] describe another technique for loop fusion based on a maxflow algorithm. The technique requires its input loop nests to be identically controlled, and it does not perform loop reversal nor interchange to enable additional fusion. Furthermore, it is unclear what the algorithm does when a potentially contractible array is consumed by multiple loop nests. Our collective scheme performs reversal, interchange and fusion simultaneously to enable contraction.

Carr and Kennedy recognized the importance of keeping array values in scalars through *scalar replacement* [CK94], which is similar to array contraction in that some array references become scalar references, but array allocation is not eliminated (*i.e.*, memory usage is not reduced). Their focus is in recognizing the opportunity in a scalar loop nest, while ours is in enabling the opportunity in an array language compiler via statement fusion.

Many techniques for improving locality by loop transformations have appeared in the literature [CMT94, KM92, MA97, WL91]. Much of this work addresses the issue of managing the conflicting goals of improving locality without sacrificing parallelism. This is a far less important issue in an array language compiler, for the compiler can assume that only the loops that it generates need to be parallelized; user loops can remain sequential. Here, we have assumed that all dimensions of all arrays are distributed and are a potential source of parallelism.

Presentations of the conventional use and construction of dependence graphs, upward exposed uses, live variable information, *etc.* appear in the compiler canon [ASU88, Muc97, Wol96].

### *4.7 Summary*

We have presented the problem of unnecessary arrays in array languages, offered an array-level solution via statement fusion and array contraction, and given a comparative evaluation of it. We find that the problem significantly impacts performance, yet existing compilations systems generally

fail to address it. We define novel machinery to enable our array-level solution, and we demonstrate that it is not only effective but that it also permits integration with other array-level optimizations, such as communication pipelining.

Despite the failings of existing compilation systems, it is clear that the unnecessary array problem is a tractable one that may be reliably solved by a compiler on the programmer's behalf. The next chapter considers a problem that cannot practically be solved entirely by the compiler, inspiring new language abstractions.

# Chapter 5

# PIPELINED PARALLEL WAVEFRONT COMPUTATIONS

Despite the successes of the previous chapter, it is impractical to expect the compiler to derive high quality code from *any* program representation. This chapter describes wavefront computations and their pipelined parallel implementation and argues that carefully chosen language abstractions greatly increase the likelihood of effective parallel implementation by the compiler without unduly increasing the programmer's burden.

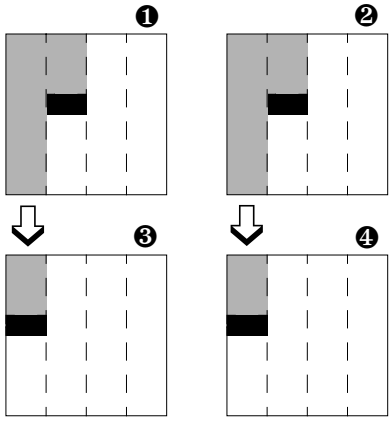
## *5.1 Motivation*

Wavefront computations are characterized by a data dependent flow of computation across a data space. The value computed in each iteration is a function of values computed in previous iterations, as in the code fragment of Figure 5.1(a). Wavefronts are common, and the scientific applications in which they appear often demand parallel execution [KBA92, SSV99]. Although the dependences they contain imply serialization, it is well known that wavefront computations admit efficient, parallel implementation via pipelining [Cyt86, Wol96]. Each processor of a pipelined implementation computes partial—rather than complete—results before sending the results on to dependent neighboring processors, as illustrated by the array distributed across four processors in Figure 5.1(b), exploiting parallelism along the direction of the wavefront. Nevertheless, the question of how wavefront computations are best presented to the compiler for the effective generation of pipelined parallel code remains open.

This chapter introduces an A–ZPL language abstraction for representing wavefront computations and compares it to other representations: programmer-implemented via message passing and compiler-discovered via automatic parallelization. The general parallel programming implications of all three approaches are well known, but not in the context of wavefront computations. They are all potentially efficient, but each has a downside. Message passing programming requires consid-

```
do j = 2, m
   do i = 2, n
      a(i,j)=(a(i,j)+a(i-1,j))/2.0
   enddo
enddo
```

(a)




(b)

Figure 5.1: (a) A simple wavefront computation and (b) its pipelined parallel implementation on four processors.

erable expertise and time to develop, debug, and tune. The benefits of automatic parallelization are only realized when a program is written in terms that the compiler is able to parallelize. And high-level parallel languages only benefit those who are willing to learn them. We assess these issues in the context of pipelining wavefront computations, and argue that A–ZPL offers a highly effective solution.

We evaluate the three approaches by using each to develop four wavefront kernels (Figure 5.2) on two dissimilar parallel machines (the IBM SP-2 and Cray T3E). The kernels are representative of a large class of wavefronts (*e.g.*, those in SWEEP3D [Accb], SIMPLE[CHL78], and Tomcatv [Sta]), and they are sufficiently simple that they allow us to focus on the first-order implications of their parallelization. We use the Message Passing Interface (MPI) [SOHL+98] as an illustration of message passing, and High Performance Fortran (HPF) [Hig97][1] of automatic parallelization.

This work provides a quantitative and qualitative assessment of developing parallel wavefront computations by the three approaches. Furthermore, we compare the development experience and performance of these approaches via a common set of kernels. The evidence we gather both confirms widely held beliefs about these representations and challenges conventional wisdom. We find that

[1]HPF is not strictly an automatically parallelized language, but it lacks intrinsic or annotational support for pipelining, relegating pipelining to an automatic parallelization/optimization task.

```
do j = 2, m
   do i = 2, n
      a(i,j) = (a(i,j)+
                a(i-1,j))/2.0
   enddo
enddo
```

(a) WF/1D/VERT

```
do j = 2, m
   do i = 2, n
      a(i,j) = (a(i,j)+
                a(i,j-1))/2.0
   enddo
enddo
```

(b) WF/1D/HOR

```
do j = 2, m
   do i = 2, n
      a(i,j) = (a(i,j)+
                a(i-1,j)+
                a(i,j-1))/3.0
   enddo
enddo
```

(c) WF/2D

```
do j = 2, m
   do i = 2, n
      a(i,j) = (a(i,j)+
                a(i-1,j))/2.0
   enddo
enddo
do j = 2, m
   do i = 2, n
      a(i,j) = (a(i,j)+
                a(i,j-1))/2.0
   enddo
enddo
```

(d) WF/1D/BOTH

Figure 5.2: Wavefront kernel computations.

the A–ZPL language-level representation is both simple to develop and consistently efficient. In addition, our study reveals surprising characteristics of commercial HPF compilers.

This chapter is organized as follows. The next section describes the representations that we consider and summarizes our experiences with their use. Section 5.3 describes the compile-time and run-time implementations of the relevant features of A–ZPL, and Section 5.4 presents performance data for each representation. The final two sections present related work and summarize.

## *5.2 Representing Wavefronts for Parallel Execution*

This section summarizes the three parallel wavefront representations we consider. Because MPI and HPF are well known, we only address the A–ZPL representation in detail. In addition, we describe our experiences using each representation to implement the kernel computations in Figure 5.2.

### *5.2.1 MPI*

Although often efficient, message passing—in this case MPI—programs are laborious to develop, for the programmer must manage every detail of parallel implementation. This is illustrated by the 626-line kernel of the ASCI SWEEP3D benchmark [Accb], only 179 lines of which are fundamental to the computation.[2] The remainder manage the complexities of implementing pipelining via message passing. Furthermore, by obscuring the true logic of a program, complexity hinders maintenance and modification. Conceptually small changes may result in substantially different implementations.

A message passing implementation of pipelining is conceptually simple, but it is surprisingly complex and difficult to develop in practice. As an illustration, the C+MPI WF/2D kernel is 40 lines long, which is large when compared to the single loop nest that it implements. In addition, its development, debugging, and performance tuning consumed three hours, a long time for such a trivial computation. Naturally, with message passing even moderate computations will be lengthy and slow to develop.

Furthermore, despite the conceptual similarity between the four kernels, the four MPI implementations differ in significant ways, such as location of communication, allocation of ghost cells, and indexing. The structure of each code is closely tied to the distribution of data and the dependences that define the wavefront, so there is little code reuse between the four implementations. In addition, we are faced with the problem of finding the best tile size (*i.e.*, the granularity of the pipeline). In order to contain development time, we dispensed with a dynamic scheme [LJ99] in favor of direct experimentation for each kernel on each machine. Naturally, the results will not extend to other machines and different problem sizes.

### *5.2.2 HPF*

An HPF program is a sequential Fortran 77/90 program annotated by the programmer to guide data distribution (via `DISTRIBUTE`) and parallelization (via `INDEPENDENT`) decisions [Hig97]. The HPF standard does not include annotations to identify computations that may be pipelined, but Gupta *et al.*. indicate that the IBM xlHPF compiler for the IBM SP-2 automatically recognizes and optimizes

[2] See the next chapter for more details about SWEEP3D.

them [GMS+95]. Some forms of task-level pipelining are supported by HPF2, but no commercial compilers fully support the new standard. Furthermore, a representation of this form would look more like an MPI code, sacrificing the benefits of HPF.

HPF programmers need not manage per processor details and explicit communication. Nevertheless, they direct the compiler's parallelization via annotations. HPF lacks annotations to identify wavefront computations, so the compiler is solely responsible for recognizing and optimizing them from their scalar representations [HKT91, RP89]. We consider the Portland Group, Inc. PGHPF and IBM xlHPF compilers separately, below.

*PGHPF*

The HPF compiler from Portland Group, Inc. (PGHPF) does not perform pipelining. We determine this by examining the intermediate message passing Fortran code produced by the `-Mftn` compiler flag. The performance data will confirm this.

PGHPF strictly obeys the `INDEPENDENT` annotations, redistributing arrays before and after loop nests so that all the annotation specified parallelism is exploited. An implication of this is that parallel loops exploit parallelism—at the cost of data redistribution—even when the user specified data distribution precludes parallelism. In this way PGHPF extracts some parallelism from two of the kernels—as we will see in the section on performance evaluation—but it is not competitive with a pipelined implementation.

Another implication of strictly respecting annotations is that they must be placed very carefully. If an `INDEPENDENT` annotation is placed on the inner loop in WF/1D/HOR, the compiler will redistribute the array inside the `j` loop, resulting in performance three orders of magnitude worse than that of the loop nest in WF/1D/VERT. The programmer may interchange the two loops, making the outer loop `INDEPENDENT`, but the resulting array traversal will have poor cache performance.

While the loop nests in WF/1D/VERT, HOR, and BOTH can use redistribution to exploit parallelism, that in WF/2D can not, for it contains dependences in both dimensions. Only pipelining will extract parallelism from this code. We found that because the loop contains no `INDEPENDENT` annotations, every array element read is potentially transmitted in the inner loop. It appears that only the source and destination processors of each scalar communication wait while the communication

takes place; thus, other processors are permitted to compute ahead, limited only by data dependences. This realizes a crude form of fine grain pipelining when arrays happen to be traversed in the right way. Despite this, the inner loop communication prevents this code from being competitive with a true pipelined implementation.

*XLHPF*

A published report indicates that IBM xlHPF performs pipelining [GMS+95]. The compiler does not provide an option for viewing the intermediate message passing code and the parallelization summary excludes this information, so we experimentally confirm that the compiler does indeed perform pipelining. Specifically, we observe that an HPF wavefront computation has single node performance comparable to the equivalent Fortran 77 program and that it achieves speedup beyond this for multiple processors.

Unlike PGHPF, xlHPF only exploits parallelism on `INDEPENDENT` loops that iterate over a distributed dimension. This fact and the pipelining optimization result in good parallel performance for all of the kernels. Despite this, we find that the pipelining optimization fails on even modestly more complex wavefronts. For example, loops that iterate from high to low indices or contain non-perfectly nested loops are not pipelined. Certainly, they could be. But the lesson is that when optimizing arbitrary code, certain cases or idioms may easily be overlooked. Conversely, a language-level solution makes explicit both the semantic and performance implications of a computation.

### *5.2.3 A–ZPL*

Here, we augment the A–ZPL discussion of Chapter 2, introducing high-level language features for pipelining wavefront computations.

*Semantics*

Array language semantics dictate that the right-hand side of an array statement must be evaluated before the result is assigned to the left-hand side. As a result, an array language compiler will not generate a loop that carries a true data dependence to a non-lexically forward statement (*i.e.*,

```
[2..n,1..n] a := 2 * a@north;
```

(a)

**for** $i \leftarrow n$ downto 2 **do**
  **for** $j \leftarrow 1$ to $n$ **do**
    $a_{i,j} \leftarrow 2 * a_{i-1,j}$

(b)

a =

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 |

(c)

```
[2..n,1..n] a := 2 * a'@north;
```

(d)

**for** $i \leftarrow 2$ to $n$ **do**
  **for** $j \leftarrow 1$ to $n$ **do**
    $a_{i,j} \leftarrow 2 * a_{i-1,j}$

(e)

a =

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 2 |
| 4 | 4 | 4 | 4 | 4 |
| 8 | 8 | 8 | 8 | 8 |
| 16 | 16 | 16 | 16 | 16 |

(f)

Figure 5.3: A–ZPL array statements (a and d) and the corresponding loop nests (b and e) that implement them. The arrays in (c and f) illustrate the result of the computations if array a initially contains all 1s.

a dependence from a statement to itself or to a preceding statement). For example, the A–ZPL statement in Figure 5.3(a) is implemented by the loop nest in 5.3(b). The compiler determines that the *i*-loop must iterate from high to low indices in order to ensure that the loop does not carry a true data dependence. If array a contains all 1s before the statement in 5.3(a) executes, it will have the values in Figure 5.3(c) afterward.

In wavefront computations, the programmer *needs* the compiler to generate loop nests with non-lexically forward loop-carried true data dependences. This may be achieved via *partial scalarization* in which the programmer explicitly inserts a loop nest to carry the dependence at the source level, as in the code of Figure 5.4(a). Unfortunately, this sacrifices the benefits of the array language representation and relegates the task of pipelining to automatic parallelization, which we address above.

Instead, we introduce a new operator, called the *prime* operator, that allows a programmer to reference values written in previous iterations of the loop nest that implements the statement containing the primed reference. Thus, the prime operator has the effect of constraining the structure of loop nests. For example, the A–ZPL statement in Figure 5.3(d) is implemented by the loop nest in 5.3(e). In this case, the compiler must ensure that a loop-carried true data dependence exists due to array a; thus, the *i*-loop iterates from low to high indices. If array a contains all 1s before

```
          for j := 2 to n do
[j,2..n]    begin
              r=aa*d@north;
              d=1.0/(dd-aa@north*r);
              rx=rx-rx@north*r;
              ry=ry-ry@north*r;
            end;
          end;
```

(a)

```
[2..n,2..n] scan
              r=aa*d'@north;
              d=1.0/(dd-aa@north*r);
              rx=rx-rx'@north*r;
              ry=ry-ry'@north*r;
            end;
```

(b)

Figure 5.4: A–ZPL representations of fragments from SPEC Tomcatv. (a) Using an explicit loop to express the wavefront. (b) Using a scan block and the prime operator. Arrays `r`, `aa`, `d`, `dd`, `rx` and `ry` are all $n \times n$.

the statement in 5.3(d) executes, it will have the values in Figure 5.3(f) afterward. In general, the directions on the primed array references define the orientation of the wavefront.

The prime operator alone cannot represent wavefronts such as the Tomcatv code fragment in Figure 5.4(a), because it only permits loop-carried true dependences from a statement to itself. We introduce a new compound statement, called a *scan block*, to allow multiple statements to participate in a wavefront computation. Primed array references in a scan block refer to values written by any statement in the block, not just the statement that contains it. For example, the A–ZPL code fragment in Figure 5.4(b) uses a scan block and the prime operator to realize the computation in Figure 5.4(a) without an explicit loop. The array reference `d'@north` refers to values from the previous iteration of the loop that iterates over the first dimension. Thus, the primed `@north` references imply a wavefront that travels from north to south. Just as in existing array languages, a non-primed reference refers to values written by lexically preceding statements, within or outside the scan block.

The notation may at first appear awkward. It is important to note, however, that experienced A–ZPL programmers are already well accustomed to manipulating arrays atomically and shifting them with the @-operator. They must only learn the prime operator, which is motivated by mathematical convention where successor values are primed. In the same vein, array languages such as Fortran 90 can be extended to include the prime operator.

The primed array references we have looked at thus far have been single cardinal directions (*i.e.*, directions in which the offset for only one dimension is nonzero, for example, representing north, south, east or west). When non-cardinal directions or combinations of different directions appear with primed references, the character of the wavefront is less obvious, but the meaning of the code is still clear. The performance implications of such codes are considered below.

*Legality*

There are a number of statically checked legality conditions: *(i)* Primed arrays in a scan block must also be defined in the block; *(ii)* the directions on primed references may not over-constrain the wavefront, as discussed below; *(iii)* all statements in a scan block must have the same rank (*i.e.*, they are implemented by a loop nest of the same depth)—this precludes the inclusion of scalar assignment in a scan block; *(iv)* all statements in a scan block must be covered by the same region; and *(v)* parallel operators' operands other than the shift operator may not be primed; this is essential because array operators are pulled out of the scan block during compilation.

An over-constrained scan block is one for which a loop nest can not be created that respects the dependences arising from the at-references. For example, primed `@north` and `@south` references over-constrain the scan block because they imply both north-to-south and south-to-north wavefronts, which are contradictory. Section 5.3 further restricts legality for performance reasons.

*Performance Model Implications*

Recall that the A–ZPL performance model allows programmers to reason about the coarse parallel performance of their codes without actually executing them. Most importantly, the A–ZPL performance model dictates that parallelism is implicit in operations on arrays. It is for this reason that the scalarized Tomcatv code fragment in Figure 5.4(a) is unsatisfactory. The performance model

tells us that each row operation will be parallel, but no parallelism exists between rows. The primed array reference solution in Figure 5.4(b) does not suffer from this problem, for the code—like most A–ZPL statements—operates on whole arrays and the degree of parallelism is principally limited by the extent of the operation. The programmer is assured that the compiler will exploit all the parallelism contained in the code.

Just as with other A–ZPL array operators, programmers may wish to reason more precisely about wavefront performance. In order to do this, the programmer must classify—paralleling the function of the compiler—each dimension of a scan block's data space as one of the following: fully parallel, sequential, or pipelined parallel. Fully parallel dimensions permit a fully parallel implementation along that dimension. Sequential dimensions must be executed according to some serial order. And pipelined parallel dimensions permit pipelining along that dimension as in Figure 5.1(b).

In most cases classifying each dimension is trivial. The programmer need only examine the directions associated with primed array references of a scan block. For example, suppose that corresponding direction entries are all nonnegative or non-positive. Clearly, any dimension for which corresponding direction entries are zero is fully parallel, because the loop that will eventually iterate over this scan block will not carry any dependences. We want at least one parallel dimension, so that pipeline can do work along that dimension as the outer loop of the pipeline. If a fully parallel dimension does not exist, we select one of the sequential dimensions for this purpose. Specifically, in this case we choose the outermost non-fully parallel dimension for cache-performance reasons. The remaining dimensions are pipelined parallel.

Below, we will use the following code fragment with different direction instantiations to illustrate how programmers may reason about their wavefront computations.

```
a := (a'@d1 + a'@d2)/2.0;
```

*Example 1:* Let `d1=d2=`$(-1,0)$. The second dimension is fully parallel, so the first dimension is pipelined parallel.

*Example 2:* Let `d1=`$(-1,0)$ and `d2=`$(0,-1)$. Neither of the dimensions are fully parallel, so the first dimension is sequential and the second is pipelined parallel. If the first dimension is distributed, the processors across which it is distributed will be wasted.

*Example 3:* Let `d1`=$(0,-1)$ and `d2`=$(0,1)$. This code is clearly over-constrained, and it will produce a compile-time flag indicating this.

*Summary*

We have presented a simple array language extension that permits the expression of loop-carried true data dependences, for use in pipelined wavefront computations. It is simple for programmers to reason about the semantics, legality and parallel implications of their wavefront code.

Contrast this with an optimization-based approach, where the programmer must be aware of the compiler's optimization strategy in order to reason about a code's potential parallel performance. Without this knowledge, the programmer is poorly equipped to make design decisions. For example, suppose a programmer writes a code that performs both north-south and east-west wavefronts. The programmer may opt to distribute only one dimension and perform a transposition between each north-south and east-west wavefront, eliminating the need for pipelining. This is likely to be much slower than a fully pipelined solution, guaranteed by our language-level approach.

### *5.3 Implementation*

This section describes our pipelined parallel implementation of primed array references and scan blocks in the A–ZPL compiler. This discussion builds on the structures and terminology used in Chapter 4 to implement statement fusion and array contraction. Below, we consider loop generation and communication generation.

Array statements containing primed array references are internally represented with a multi-loop structure just like any other array statement. Scan blocks require special consideration because they must be implemented by a single loop nest, so each scan block is transformed into a single multi-loop, rather than a sequence of multi-loops.

Unconstrained distance vectors (UDVs) are constructed as described in the last chapter with one exception. The prime operator transforms what an array language would otherwise interpret as an anti-dependence into a true dependence. In order to achieve this, the unconstrained distance vectors associated with primed array references are simply negated. For example, the statement `A:=A@(0,1)` gives rise to an anti-dependence with the UDV (0,1), while the statement

`A:=A'@(0,1)` gives rise to a flow dependence with the UDV (0,-1). These unconstrained distance vectors and the accompanying array statement dependence graph are used for statement fusion and array contraction as we have already described. Thus, scan blocks are optimized just like all A–ZPL array statements.

Tiling is essential for balancing parallelism and communication overhead. Tiling is inhibited—without data redistribution—when a dimension is distributed and there are certain *conflicting entries* (*i.e.*, some positive and negative) in the UDVs associated with primed array references. This is a result of the fact that data must be transmitted back and forth across the processor boundary of this dimension, preventing the aggregation of communication. For example, there are conflicting entries in the second dimension of UDVs $(-1,1)$ and $(-1,-1)$; thus the first dimension can not be tiled to reduce communication overhead when pipelining the second dimension. Because of this and the fact that all dimensions are potentially distributed, we require that within a scan block all UDVs associated with primed references must contain no conflicting entries.

Recall that each dimension is either fully parallel, sequential, or pipelined parallel, as described in Section 5.2.3. Tiling is used in the sequential and parallel dimensions to create pipeline stages. The pipelined dimensions are not tiled. The dimensions that are to be tiled are tagged as such in the multi-loop structure. At code generation time, outer loops are generated for the dimensions that are to be tiled, and inner loops are generated to iterate over a single tile.

Tile size determines performance, balancing parallelism and communication overhead. Specifically, smaller tiles increase parallelism but increase communication overhead. The relative cost (in time) of communication and the wavefront computation determine the appropriate tile size. Numerous models and dynamic schemes have been developed to calculate tile size [OSKO95, ABR96, DRR96, AR97, ARY98, LJ99], and this is an important component of a language-level solution to pipelining wavefront computations. Nevertheless, we have not fully implemented this component in the A–ZPL compiler. In the interim, we manually select tile size at run-time.

The A–ZPL compiler inserts communication for nonlocal references, such as those implied by the at operator [CS97, Cho99]. The compiler performs a backward traversal of the control flow graph, inserting communication primitives for the at-references it encounters. Effectively, a receive is inserted just prior to an at-reference, and a send is inserted just after the preceding definition of

the referenced array, overlapping communication and computation.[3] This scheme is only slightly modified for primed at-references. First, the communication associated with primed at-references is associated with the scan block that contains the reference rather than moving them around to overlap communication and computation. Second, receives appear at the beginning of the scan block and sends appear at the end. At code generation time, these communication primitives appear within the outer tile loops and outside the inner tile loops; thus, data is sent and received at the granularity of the tile.

In summary, the implementation of tiling is very simple, leveraging the existing structures in the A–ZPL compiler.

### *5.4 Performance*

We gather performance data on two parallel machines: a 272-processor Cray T3E-900 (450MHz DEC Alpha 21164 nodes) and 192-processor IBM SP-2 (160MHz Power2 Super Chip nodes). We use a number of compilers in this evaluation: on the T3E, we use the Cray CF90 Version 3.2.0.1 and Portland Group, Inc. PGHPF v2.4-4; on the SP-2, we use IBM xlf Fortran v4.1.0.6, IBM xlHPF v1.4, IBM xlc v3.1.4.0, and PGHPF v2.1. On both machines we use the University of Washington zc v1.15 A–ZPL compiler [ZPL]. All compilers are used with the highest optimization level that guarantees the preservation of semantics.

We study four different representations: C+MPI, A–ZPL, xlHPF, PGHPF. The C+MPI code is a well-tuned pipelined message-passing program. It represents the (practical) best that can be achieved on these machines using the C programming language. Because the xlHPF compiler is only available on the SP-2, we do not have results for it on the T3E.

We use the codes in Figure 5.2 for this evaluation. For all the experiments, the `a` array is distributed across a dimension that gives rise to a loop-carried dependence (*e.g.*, the first dimension in WF/1D/VERT) so as to isolate the impact of pipelining. Although it appears that WF/1D/VERT and WF/1D/HOR do not require pipelining (*i.e.*, there exists a distribution that permits complete parallel execution), these kernels may appear in a context that requires a different, less favorable

[3] A–ZPL actually uses a more portable communication interface than that of send/receive [CCS98], but the simplification is sufficiently accurate for this discussion.

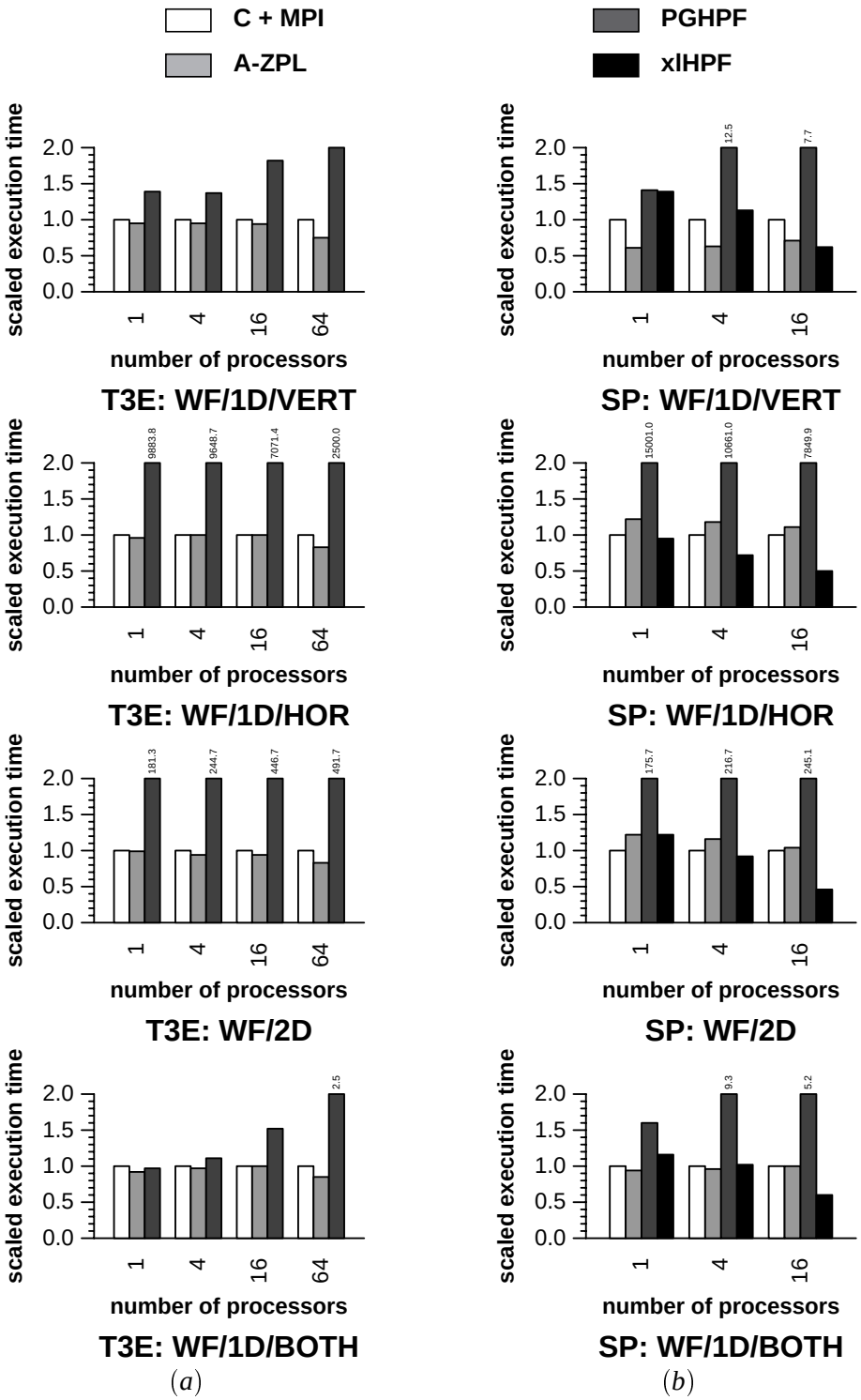
distribution.

We find that the single processor execution times for C+MPI and A–ZPL—which generates C code—are comparable to that of a sequential C program. Similarly, on a single processor, xlHPF and PGHPF match sequential Fortran. On the T3E, sequential C code typically executes in twice the time of comparable Fortran codes, while on the SP-2 this ratio varies with the character of the kernel. Such disparities between C and Fortran implementations of the same computation are common. In any case, it is clear that the observed scaling behavior is relative to an efficient baseline.

All the performance data is summarized by the graphs in Figure 5.5. These graphs depict execution time, so lower bars indicate better performance. Furthermore, the performance is scaled relative to C+MPI. First, observe that the A–ZPL performance keeps pace with that of C+MPI. This indicates that A–ZPL is both performing as well and scaling as well as the hand coded program. At times the A–ZPL code even surpasses C+MPI, because it performs low level optimizations for more efficient array access. Consider the PGHPF performance. It is competitive on a single processor for the WF/1D/VERT and WF/1D/BOTH kernels, but it quickly trails off as the number of processors increases. This is because, PGHPF redistributes the data to achieve parallelism, which does not scale. The SP-2 exaggerates this effect, because its high communication costs outweigh the benefits of redistribution. Furthermore, for WF/1D/HOR and WF/2D significant communication appears in the inner loop, resulting in abysmal performance (the bars are off the graph!).

XlHPF is competitive with the C+MPI and A–ZPL, because it performs pipelining. The single processor bars highlight disparities in local computation performance. A–ZPL performs considerably better than any of the others for WF/1D/VERT. We hypothesize that the dependences in this kernel thwart proper array access optimization by the xl optimizer (used by both the Fortran and C compilers). The A–ZPL code does not suffer from this, because its compiler generates direct pointer references rather than using C arrays. When the C+MPI code is modified in this way, its performance matches A–ZPL. Conversely, A–ZPL is worse for WF/1D/HOR. Again, we believe this is an optimization issue. When the A–ZPL code is modified to use C arrays rather than pointer manipulation, it matches HPF. The summary is that when we ignore the differences that arise from using C versus Fortran, the C+MPI, xlHPF, and A–ZPL kernel performance are comparable.

One might conclude that automatic parallelization (*i.e.*, HPF) is the solution and one should use the xlHPF compiler. After all, for the evaluation codes it performs very well. This is a poor




(a) (b)

Figure 5.5: Pipelining performance summary. Kernel names are from Figure 5.2. Note that many of the PGHPF bars are off the scale of these graphs.

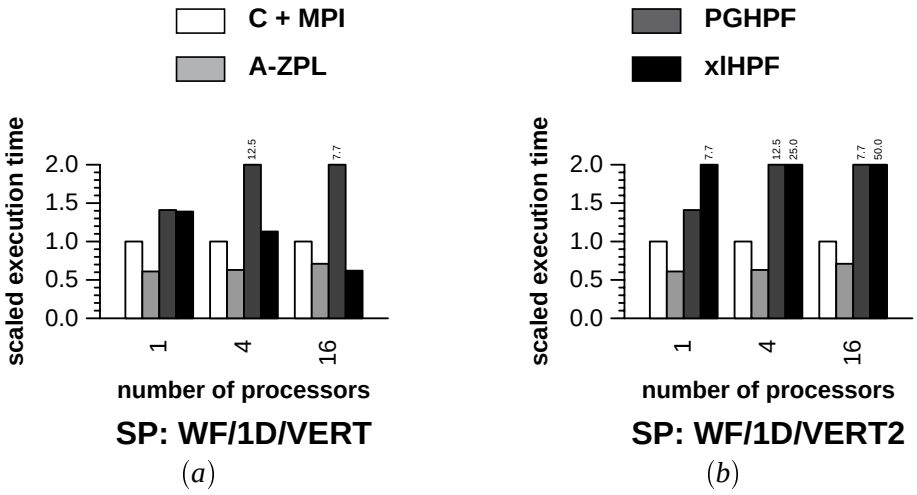



Figure 5.6: Mare pipelined performance data. The graph in (a) is for kernel WF/1D/VERT, while that in (b) is for the same code except that loops iterate from high to low indices. Note that the xlHPF performance goes from very good to very bad.

conclusion for two reasons. First, the xlHPF compiler is only available on IBM machines; thus, any codes written on IBM machines will not be portable. And second, the xlHPF compiler fails to pipeline all but the simplest of codes. For example, imperfectly nested loops and control flow inhibit this transformation. Sensitivity to iteration direction is a particularly egregious shortcoming. For example, if we iterate from high to low indices in the WF/1D/VERT kernel, the xlHPF execution time balloons, as in Figure 5.6(b). The xlHPF compiler very effectively pipelines simple codes, but its fragility and lack of portability make it an impractical representation for serious codes.

### *5.5 Related Work*

In a study of parallel performance models, Ton Ngo first suggested the value of language-level support for pipelining wavefront computations [Ngo97]. This work validates this insight in the context of a programming language with a well-defined performance model.

It is well known that wavefront codes admit pipelined parallel implementations. Cytron [Cyt86] and Rogers and Pingali [RP89] describe early experiences doing just this, and many others have further developed the technique [HKT91, HKT92, SY91, CTY94, AWMC+95, BL99], in particular how a compiler can automatically pipeline scalar code (*e.g.*, in the context of HPF). In addi-

tion, considerable applied and theoretical effort has been devoted to discovering the tile size that optimizes parallelism. Both static [OSKO95, ABR96, DRR96, AR97, ARY98] and dynamic approaches [LJ99] have been explored. Nevertheless, we are aware of no work that considers the parallel performance implications of language-level support for pipelining wavefront computations. Aside from the automatic parallelization work, all the preceding work can be applied to A–ZPL implementation of pipelining to improve its performance.

### *5.6 Summary*

We have considered several program representations for wavefront computations. The message passing representation performs well, but it sorely lacks usability. The programmer must write an unnecessarily long and complex message passing program for even the simplest computations. The automatically parallelized sequential representation is a very simple representation, but its performance is closely tied to the machine on which it is run, the compiler that is used, and the idiosyncrasies of the code. In general, the programmer can not be assured that the compiler will generate an efficient pipelined parallel implementation.

Our experiments suggest that it is impractical for compiler analysis alone to derive an efficient pipelined parallel implementation of wavefront computations. Accordingly, we have introduced novel language abstractions, that unambiguously identify the parallelism in wavefront computations; and we have quantitatively and qualitatively compared them to the alternatives. We show that of the alternatives considered, only A–ZPL provides both performance and usability.

# Chapter 6

# A CASE STUDY: SWEEP3D

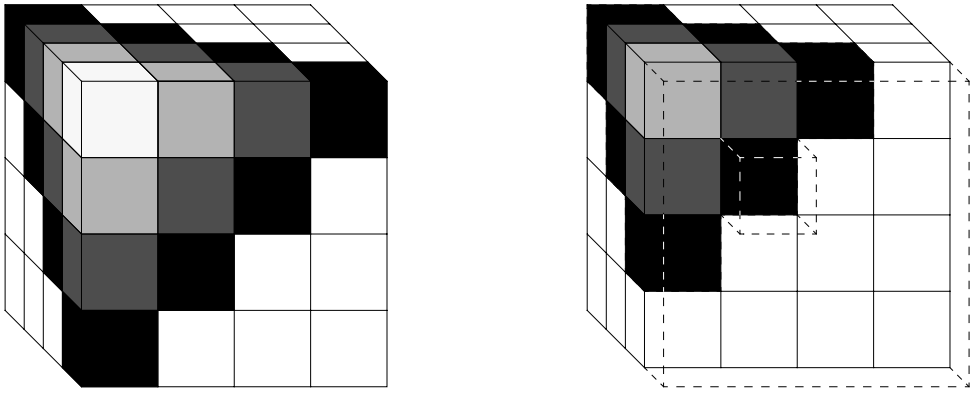
This chapter evaluates the A–ZPL design and implementation in the context of a particular application, the SWEEP3D benchmark from ASCI. First, we describe the abstract characteristics of the application; then we consider particular implementations in message passing Fortran and A–ZPL. Finally, we qualitatively and quantitatively compare these two representations of the SWEEP3D computation.

## *6.1 SWEEP3D Overview*

The SWEEP3D benchmark comes from the Accelerated Strategic Computing Initiative (ASCI) [Acca, Accb]. ASCI is a collaboration among the national labs to advance defense-oriented computer modeling and simulation, including work in hardware, software, and supportive environments. The SWEEP3D benchmark is intended to represent complete, ASCI-caliber applications in order to evaluate parallel machines. We have chosen to evaluate SWEEP3D not only because it highlights the work discussed thus far in this dissertation, but also because the quality of the A–ZPL implementation is indicative of a significant class of applications, beyond those SWEEP3D models.

SWEEP3D solves a 1-group, time-independent, discrete ordinates, three-dimensional Cartesian geometry neutron transport problem. The problem space consists of three *spatial dimensions*, and one *angular dimension*, represented by an array of angles [Accb]. All the illustrations in this chapter exclude the angular dimension for reasons of simplicity and clarity.

Each cell in the three-dimensional space contains four equations in seven unknowns—six faces plus the cell center—and boundary conditions complete the system of equations. A direct ordered solver produces a *sweep* as follows. For each cell, three known spatial inflows, originating at boundaries, are used to calculate the cell center and three outflows, providing the inflows for three adjacent cells. A cell cannot compute its outflows before its inflows become available, so a wavefront



(a) (b)

Figure 6.1: A snapshot of a SWEEP3D wavefront traveling across the three spatial dimensions, from the upper left-hand corner of the front face to the lower right-hand corner of the back face. The white cells have not yet been computed, the black cells are the most recently computed, and the gray cell have been computed in the past (lighter cells computed in the more distant past). External and cross section views appear in (a) and (b), respectively.

of computation travels across the problem space with a recursion dependence in each of the spatial dimensions [Accb]. The core of SWEEP3D contains eight sweeps, one originating at each corner of the cube.

Figure 6.1 illustrates a SWEEP3D wavefront. The figure represents a snapshot of a $4^3$ grid of cells. The white cells have not yet been computed, the black cells have most recently been computed, and the gray cell have been computed in the past (lighter cells indicating the more distant past). We assume that the north, west, and front face boundaries are known, so computation begins with the cell at the upper left-hand corner of the front face. The wave spreads out from there as it travels to the lower right-hand corner of the back face, as shown in Figure 6.1(a). Figure 6.1(b) shows a cross section of the same grid at the same instant in time. The interior black cell may only be computed when its north, west, and front (represented by the box) neighbors have been computed.

An instance of the problem solved by SWEEP3D is defined by the following parameters. $IT_G$, $JT_G$, $KT_G$, and $MM$ are the sizes of the three spatial dimensions and the angular dimension, respectively; and $NM$ is the number of flux & source angular moments, typically small. The parallel implementation described below has the following additional parameters. $NPE_i$ and $NPE_j$ are the number

of processors across which the $i$ and $j$ dimensions are distributed. $IT$ and $JT$ are the size along the $i$ and $j$ dimensions, respectively, for one processor (*i.e.*, $IT = IT_G \div NPE_i$ and $JT = JT_G \div NPE_j$). For simplicity, let $KT = KT_G$. $MK$ and $MMI$ are the size of a tile along the $k$ spatial and angular dimensions. $KB$ and $MMO$ give the number of tiles along the same two dimensions (*i.e.*, $KB = KT_G \div MK$ and $MMO = MM \div MMI$). Thus, each processor has $KB \times MMO$ tiles, each of size $MK \times MMI$.

SWEEP3D is best parallelized via pipelining as follows. The $i$ and $j$ spatial dimensions are block distributed. Rather than each processor waiting for all the inflows, computing all its cells, and transmitting all its outflows, they only compute a subset of the planes in the $k$ dimension—called $k$ planes—between communication steps, creating a pipeline. In addition, only a subset of the angles per plane are computed at a time, useful in further adjusting the granularity of the pipeline [KBA92].

Figure 6.2 illustrates a pipeline parallel implementation of a SWEEP3D wavefront. Like Figure 6.1 this figure is a snapshot of the computation over a three-dimensional grid of cells. Unlike Figure 6.1, the $i$ and $j$ dimensions of a $12 \times 12 \times 4$ array ($IT_G = JT_G = 12$ and $KT_G = 4$) are distributed across a $3 \times 3$ processor grid ($NPE_i = NPE_j = 3$), so that each processor contains $4^3$ cells ($IT = JT = KT = 4$). Again, white cells have yet to be computed, black cells are most recently computed, and gray cells have been computed in the past (lighter cells in the more distant past). Before beginning work on the next k-plane, each processor sends partial results to its east and south neighbors, forming a diagonal pipeline across the two-dimensional processor space. Here, we ignore the angular dimension, but $MK = 1$ and $KB = 4$.

Figure 6.2 illustrates the potential for inter-sweep parallelism. Each sweep step—save the last—is followed by another sweep typically in an orthogonal direction, permitting one sweep to start before the previous finishes. For example, suppose that the northwest-to-southeast wavefront in Figure 6.2 will be followed by another traveling from southwest-to-northeast. The southwest processor will finish its contribution to the first sweep and begin the second, while the three southeast processors are still working on the first. In addition, the code is able to exploit intra-tile parallelism using compilers that recognize the appropriate directives and machines that are able to exploit very fine-grain parallelism.

As a minor simplification, all codes considered here do not implement diffusion synthetic acceleration (DSA), and our example codes exclude the SWEEP3D “fix-up” step. This has no impact on the parallel implications of the code nor its general structure.

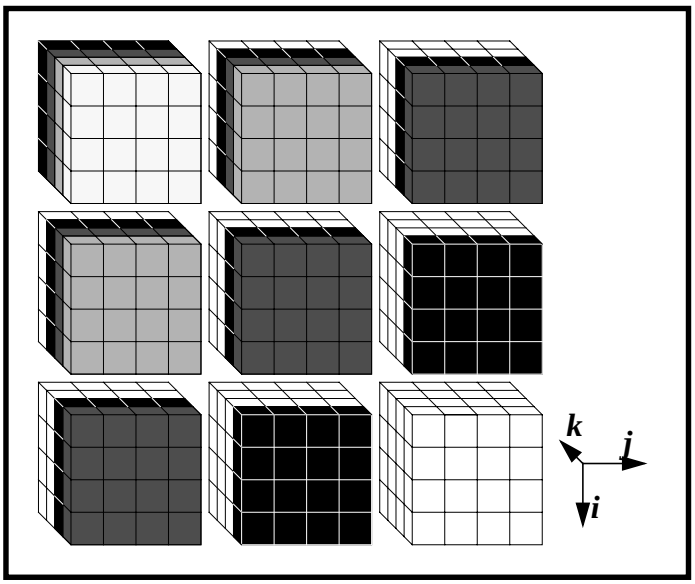



Figure 6.2: Snapshot of SWEEP3D wavefront, illustrating data distribution and pipelining. Spatial dimensions *i* and *j* are block distributed across a $3 \times 3$ processor grid. Dimension *k* is not distributed. White cells have not yet been computed and black cells are most recently computed.

## *6.2 Implementations*

### *6.2.1 Base Implementation: Message Passing Fortran 77*

ASCI provides a Fortran 77 implementation of SWEEP3D. Message passing is achieved via a library that maps to either MPI or PVM [Accb]. This code is carefully crafted to best realize the serial and parallel performance capabilities of machines. Any shortcomings that exist in this code are due to the implementation context, *not* the skills of its authors. It is implemented by 10 files, containing a total of 1346 useful (*i.e.*, non-comment and non-blank) lines of code. The main function, `sweep()`, is 292 lines long.

Figure 6.3 summarizes the 292 line `sweep()` function in 86 lines. Although most of the code has been replaced by descriptive comments, the overall structure remains. The core of the computation appears in lines 33–49, which computes a single tile in the four-dimensional iteration space. A recurrence is formed on arrays `phijb` and `phikb` and scalar `phiir`, which is eventually stored in array `phiib`. Each of these arrays lacks a dimension for the dimension that gives rise to the

Table 6.1: Arrays in Fortran SWEEP3D.

| *arrays* | *size* |
|---|---|
| `hi`, `di`, `phi` | $IT$ |
| `hj`, `dj` | $JT$ |
| `hk`, `dk` | $KT$ |
| `w`, `mu`, `eta`, `tsi`, `wmu`, `weta`, `wtsi` | $MM$ |
| `sigt`, `pflux`, `srcx` | $IT \times JT \times KT$ |
| `pn` | $MM \times NM \times 8$ |
| `phiib`† | $JT \times MK \times MMI$ |
| `phijb`† | $IT \times MK \times MMI$ |
| `phikb`† | $IT \times JT \times MMI$ |
| `src`, `flux` | $IT \times JT \times KT \times NM$ |
| `sigs` | $IT \times JT \times KT \times 2$ |

† These arrays are bound by tile size rather than problem size.

recurrence. For example, `phikb` does not have an allocated *k*-dimension, so references to the array induce a loop-carried dependence for the *k* loop. The three loops beginning at line 33 iterate over the four-dimensional iteration space in such a way that one of the loops is parallel (*i.e.*, each iteration is independent), exposing intra-tile parallelism.

The outermost loops, beginning at lines 1 and 9, iterate over the batches of angles and k-planes that form tiles. Aside from these loops, the code that precedes the core sets up the *i*, *j* and *k* inflows used by the core. Based on a processors relative position, the inflows are either 0 or values received from a neighboring processor, *i.e.*, the neighbor's outflow. Similarly, the code following the core either sends the outflows to neighbors or performs a leakage calculation.

Table 6.1 gives the size on each processor of the important arrays in the Fortran implementation of SWEEP3D. Many of the arrays are one-dimensional, three of the arrays are bound by tile size rather than problem size (`phiib`, `phijb`, and `phikb`), and only three arrays represent all three spatial dimensions (`src`, `flux`, and `sigs`), which are much larger than the angular dimension. In summary, these arrays are space efficient. We will see why this is important when we examine the A–ZPL implementation.

```
DO mo = 1, mmo  ! outer angles loop (batches of mmi angles)
   do mi = 1, mmi  ! K-inflows (k=k0 boundary)
      do j = 1, jt
         do i = 1, it
            phikb(i,j,mi) = 0.0d+0
         end do
      end do
   end do
   DO kk = 1, kb  ! outer k-planes loop (batches of mk-planes)
      if (ew_rcv .ne. 0) then  ! I-inflows for block (i=i0 boundary)
         call rcv_real(ew_rcv, phiib, nib, ew_tag, info)
      else
         do mi = 1, mmi
            do lk = 1, nk
               do j = 1, jt
                  phiib(j,lk,mi) = 0.0d+0
               end do
            end do
         end do
      endif
      if (ns_rcv .ne. 0) then  ! J-inflows for block (j=j0 boundary)
         call rcv_real(ns_rcv, phijb, njb, ns_tag, info)
      else
         do mi = 1, mmi
            do lk = 1, nk
               do i = 1, it
                  phijb(i,lk,mi) = 0.0d+0
               end do
            end do
         end do
      endif
      ! compute 1 tile/block
      DO idiag = 1, jt+nk-1+mmi-1  !JK-diagonals with MMI pipelined angles
         ! DO PARALLEL
         DO jkm = 1, ndiag
            DO i = i0, i1, i2   ! I-line recursion: without flux fixup
               ci =  mu(m)*hi(i)
               dl = ( sigt(i,j,k) + ci + cj + ck )
               dl = 1.0 / dl
               ql = phi(i) + ci*phiir + cj*phijb(i,lk,mi) + ck*phikb(i,j,mi)
               phi(i) = ql * dl
               phiir          = 2.0d+0*phi(i) - phiir
               phii(i)        = phiir
               phijb(i,lk,mi) = 2.0d+0*phi(i) - phijb(i,lk,mi)
               phikb(i,j,mi)  = 2.0d+0*phi(i) - phikb(i,j,mi)
            END DO ! i
            phiib(j,lk,mi) = phiir
         END DO ! jkm
      END DO ! idiag
      if (ew_snd .ne. 0) then  ! block I-outflows (i=i1 boundary)
          call snd_real(ew_snd, phiib, nib, ew_tag, info)
      else
          leak = 0.0
          do mi = 1, mmi
             do lk = 1, nk
                do j = 1, jt
                   leak = leak + wmu(m)*phiib(j,lk,mi)*dj(j)*dk(k)
                end do
             end do
          end do
          leakage(1+i3) = leakage(1+i3) + leak
      endif
      if (ns_snd .ne. 0) then  ! block J-outflows (j=j1 boundary)
         call snd_real(ns_snd, phijb, njb, ns_tag, info)
      else
         leak = 0.0
         do mi = 1, mmi
            do lk = 1, nk
               do i = 1, it
                  leak = leak + weta(m)*phijb(i,lk,mi)*di(i)*dk(k)
               end do
            end do
         end do
         leakage(3+j3) = leakage(3+j3) + leak
      endif
   END DO ! kk
   leak = 0.0  ! K-outflows (k=k1 boundary)
   do mi = 1, mmi
      do j = 1, jt
         do i = 1, it
            leak = leak + wtsi(m)*phikb(i,j,mi)*di(i)*dj(j)
         end do
      end do
   end do
   leakage(5+k3) = leakage(5+k3) + leak
END DO ! mo
```

Figure 6.3: Greatly simplified core of SWEEP3D in Fortran.

Table 6.2: Arrays in A–ZPL SWEEP3D.

| *arrays* | *declared size* | *final size* |
|---|---|---|
| `ci`, `cj`, `ck`, `ti`, `tj`, `tk`, `fixed`, `done`, `dl`, `ql`, `phi` | $MM \times KT \times JT \times IT$ | 1 |
| `hi`, `di` | $IT$ | $IT$ |
| `hj`, `dj` | $JT$ | $JT$ |
| `hk`, `dk` | $KT$ | $KT$ |
| `w`, `mu`, `eta`, `tsi`, `wmu`, `weta`, `wtsi` | $MM$ | $MM$ |
| `sigt`, `pflux`, `srcx` | $KT \times JT \times IT$ | $KT \times JT \times IT$ |
| `pn` | $8 \times MM \times NM$ | $8 \times MM \times NM$ |
| `phiib` | $MM \times KT \times JT \times IT$ | $MM \times KT \times JT$ |
| `phijb` | $MM \times KT \times JT \times IT$ | $MM \times KT \times IT$ |
| `phikb` | $MM \times KT \times JT \times IT$ | $MM \times JT \times IT$ |
| `src`, `flux` | $NM \times KT \times JT \times IT$ | $NM \times KT \times JT \times IT$ |
| `sigs` | $2 \times KT \times JT \times IT$ | $2 \times KT \times JT \times IT$ |

### *6.2.2 A–ZPL Implementation*

A computationally equivalent A–ZPL implementation has been built, using the ASCI message passing Fortran code as a model. The A–ZPL implementation (Figure 6.4) does not use message passing, rather, pipelined parallelism is implicit in scan blocks and primed at references. In mirroring the structure of the Fortran code, the A–ZPL code is implemented by 10 files, containing a total of 419 lines of useful code. The main function, `sweep()`, is 115 lines long.

Figure 6.4 summarizes the 115 line `sweep()` function in 24 lines. This figure also includes array declarations. The core of the computation, including a scan block, is bound by lines 7 and 18. Lines 3–5 compute the boundary inflows in each dimension, and lines 20–22 compute leakage based on the corresponding boundary outflows. Note that the meaning of the dimensions of the arrays in the A–ZPL is a transposition of those of the Fortran. This reflects the difference of column-major versus row-major array allocation in the two languages.

Table 6.2 gives the size on each processor of the important arrays in the A–ZPL implementation of SWEEP3D. Two sizes are given: *(i)* as declared by the programmer and *(ii)* the final size after array contraction. The next section will discuss this data and compare it to that of the message passing Fortran implementation.

```
region BigR = [1..mm,0..n_k+1,0..n_j+1,0..n_i+1];
       R =    [1..mm,1..n_k,1..n_j,1..n_i];
       Rm =   [1..mm,*     ,*     ,*     ];
       Ri =   [*    ,*     ,*     ,1..n_i];
       Rj =   [*    ,*     ,1..n_j,*     ];
       Rk =   [*    ,1..n_k,*     ,*     ];
       Rijk = [*    ,1..n_k,1..n_j,1..n_i];

var ci, cj, ck, ti, tj,
    tk, dl, ql, phi     : [R] double;
    hi, di              : [Ri] double;
    hj, dj              : [Rj] double;
    hk, dk              : [Rk] double;
    mu, eta, tsi,
    w, wmu, weta, wtsi  : [Rm] double;
    Sigt, pflux, Srcx   : [Rijk] double;
    pn                  : array [1..8] of [Rm] array [1..nm] of double;
    phiib, phijb, phikb : [BigR] double;
    Src                 : [Rijk] array [1..nm] of double;
    flux                : [Rijk] array [1..nm] of double;
    Sigs                : [Rijk] array [1..2] of double;

[R]          begin

[lasti of R]   phiib := 0.0;  -- boundary i inflow
[lastj of R]   phijb := 0.0;  -- boundary j inflow
[lastk of R]   phikb := 0.0;  -- boundary k inflow

               ci :=  mu * hi;
               cj := eta * hj;
               ck := tsi * hk;
               dl := 1.0 / (Sigt + ci + cj + ck);

               scan
                 ql := phi + ci*phiib'@lasti + cj*phijb'@lastj + ck*phikb'@lastk;
                 phi := ql * dl;
                 phiib := 2.0*phi - phiib'@lasti;
                 phijb := 2.0*phi - phijb'@lastj;
                 phikb := 2.0*phi - phikb'@lastk;
               end;

[lasti in R]   leakage[1+i3] +=  wmu * phiib * dj * dk;  -- final i outflow
[lastj in R]   leakage[3+j3] += weta * phijb * di * dk;  -- final j outflow
[lastk in R]   leakage[5+k3] += wtsi * phikb * di * dj;  -- final k outflow

             end;
```

Figure 6.4: Core of SWEEP3D in A–ZPL.

Recall that directions in A–ZPL are static; thus, the same code can not be used to compute all eight wavefronts. Static directions are an important part of the A–ZPL design, so we are unwilling to completely forsake them, but mechanisms that solve this problem via some form of polymorphism and specialization are currently in design. In the interim, we have used the M4 macro package to stamp out code for the eight wavefronts, buffering the programmer from code replication. The macro expansion introduces a factor of eight code bloat, but the programmer need never see or manipulate this intermediate code. Interestingly, even after this gratuitous code explosion, the final A–ZPL code is still shorter than the message passing Fortran.

## *6.3 Discussion*

This section considers a number of concerns that may impact the efficacy of the implementations introduced in the previous section.

### *6.3.1 A–ZPL Concerns*

In comparing the A–ZPL and Fortran implementations of SWEEP3D, a performance-minded programmer is likely to have three main concerns. First, array languages require the expansion of what would otherwise be scalars to be full arrays (*e.g.*, `ci`, `cj`, and `ck`). Second, the programmer has no control over loop generation; can the compiler produce loop nests that compete with the carefully constructed Fortran program? And third, and perhaps most importantly, is the compiler up to the task of generating an effective pipelined parallel implementation from the given source program?

#### *Scalar Expansion*

The term *scalar expansion*, or *scalar promotion*, was first used to describe a technique (implemented either by hand or in a compiler) in which scalar references in a loop are replaced by array references in order to eliminate loop-carried data dependences on the scalar and enable the parallel execution of the loop [Wol96]. Similarly, array language semantics require that arrays referenced together must be conformable; thus, some scalar references must be expanded to arrays. Arrays `ci`, `cj`, `ck`, `ti`, `tj`, `tk`, `fixed`, `done`, `dl`, and `ql` are examples of this for the A–ZPL code in Figure 6.4.

Scalar expansion is a hindrance for a number of reasons. Most obviously, it wastes memory. In this case, what would otherwise be a scalar is a full four-dimensional array. For a moderately sized problem, this results in a factor of 10,000 increase in memory required for a single variable. Wasted memory not only limits the size of problems that may be solved in a given memory size, it degrades performance in two ways. First, array references to expanded arrays pollute the data cache and limit its efficiency. And second, scalar values are more likely to be kept in machine registers which are more quickly retrieved than even cached data in memory; this is a property of modern load-store architectures.

There a two ways to address the memory inefficiency of scalar expansion. First, under certain circumstances, a programmer may use a flood array. A flood array replicates a lower dimensional structure along one or more dimensions to create an array that can be referenced as if it were a higher dimensional structure. Flood arrays are used extensively in the A–ZPL implementation of SWEEP3D. All the non-four-dimensional arrays according to the second column of Table 6.2 use flood arrays. Arrays `hi` and `di`, for example, are referenced as if they were four-dimensional arrays but they only represent a single vector along the *i* dimension. Flood arrays do not completely solve the problem, because they can not be used in cases when the promoted dimensions do not represent replicated data. For example, array `ci` contains on arbitrary value in each element. Array contraction addresses this problem.

Array contraction is another line of defense against the memory inefficiency of scalar expansion. Array contraction gathers loop nests together so that lower dimensional arrays can be used to hold certain values. Because it is entirely the role of the compiler to generate loop nests from array statements, it is the compiler's sole responsibility to perform array contraction. Chapter 4 has shown that the compiler is able to perform contraction to the degree that the programmer may ignore this issue. The final column of Table 6.2 gives the array sizes after both full and partial array contraction have been performed. The arrays in the first line are contracted to a scalar, and `phiib`, `phijb`, and `phikb` have one dimension contracted.

Compare the final sizes of the arrays in Tables 6.1 and 6.2. The tables indicate that the arrays in Fortran and A–ZPL are the same except for arrays `phi`, `phiib`, `phijb`, and `phikb`. The Fortran array `phi` becomes a scalar in A–ZPL. We suspect that the Fortran code does not do the same in an effort to exploit the idiosyncrasies of a particular Fortran compiler. The other three arrays are smaller

in Fortran. Specifically, they are the size of a tile along the *k* and angular dimensions in Fortran, while they are the size of each processors' ownership in these dimensions in A-ZPL. Thus, the A–ZPL implementation of SWEEP3D uses somewhat more memory for these three arrays. Section 6.4 will answer the question of whether this significantly impacts performance.

Another implication of scalar expansion is that it commonly results in redundant computation. For example, array `ck` is a function of array `tsi` and `hk`, which are flooded in all but the angular and *k* dimensions, respectively. Thus, `ck` redundantly computes values in the *i* and *j* dimensions. The A–ZPL compiler optimizes this redundancy via loop-invariant code motion.

*Loop Generation and Scalar Performance*

In an array language, such as A–ZPL, the compiler is responsible for mapping the more abstract array representation to a collection of semantically equivalent loop nests. The Fortran code in Figure 6.3 illustrates the potential complexity of these loops: large, imperfectly nested, and highly optimized by the programmer. Unfortunately, a direct compilation of array statements generates a single loop nest for each statement. This is not a winning strategy.

The statement fusion strategy described in Chapter 4 fully addresses these concerns. In fact, the C code generated by the A–ZPL compiler differs from the Fortran code in only one minor way: the global boundary initializations and the leakage calculations do not appear in the core loop. It is for this reason that the A–ZPL arrays `phiib`, `phijb`, and `phikb` have a final size determined by the problem size rather than the tile size, as in the Fortran.

An important practical concern arises from A–ZPL's use of C as a target language. It is generally accepted that Fortran compilers produce higher performance code than does C for numerical applications. This state of affairs exists for several related reasons: *(i)* Fortran compiler technology is more mature than that of C, *(ii)* Fortran compilers are designed to optimize array accesses, a dominant characteristic of numerical codes, *(iii)* C programs often contain pointer manipulations that thwart analysis and optimization, and *(iv)* Fortran compilers often recognize particular programming idioms, such as matrix-vector product, and replace this code with calls to highly optimized BLAS routines. The A–ZPL compiler addresses this problem by performing a number of scalar optimizations so as not to rely on the ability of the back-end C compiler to optimize array accesses.

We will return to this issue when we examine performance data.

*Parallel Implementation*

Is the A–ZPL compiler up to the task of generating an effective parallel implementation from the given source program? Can it generate pipelined code and exploit inter-sweep parallelism? Can it exploit intra-tile parallelism like the Fortran code? Previous chapters and Section 6.4 answer the first question. The answer to the second question is, "yes." The processors of an A–ZPL application are only loosely synchronized; thus, one can start the next sweep before others have finished the previous one. As for the third question, the answer is, "no." The current A–ZPL compiler does not attempt to exploit this kind of multi-resolution parallelism.

### *6.3.2 Message Passing Fortran Concerns*

The message passing Fortran implementation of SWEEP3D was designed to be fast, and it has been evaluated on several machines and demonstrated to achieve this goal [CKPN00, LHZ98, SSV99]. Thus, we present only a few performance concerns. Instead, we focus on the practicality of using message passing Fortran of a similar system for implementing codes like this.

*Portable Performance*

Any message passing implementation is susceptible to problems of portability, for it fixes a particular communication mechanism: message passing. Machines that support other communication mechanisms may be underutilized by message passing. For example, the Cray T3E supports low latency one-sided communication, so programs customized for message passing do not fully exploit the machine. In the specific case of SWEEP3D, communication represents a small portion of the total execution time, and we expect that this issue of portability will be minor. But not all message passing applications are so lucky.

*Complexity*

We now turn to the issue of practicality. The reader is no doubt struck by the disparity in apparent complexity of the Fortran and A–ZPL implementations of SWEEP3D, highlighted by the differ-

ence in length, 292 versus 115 lines. The source of the problem comes from the fact that Fortran does not provide abstractions for parallel programming. This is not a deficiency of Fortran—a serial language—rather, it is a deficiency of using a serial language to write parallel programs. The program becomes unnecessarily mired in the implementation details of simple abstract ideas, in this case a pipelined wavefront computation. Similarly, before high-level serial programming languages, programmers were burdened with explicitly managing a call stack.

Unnecessary asymmetry is an example of how complexity limits the practicality of this code. Specifically, exactly the same computation takes place in the three spatial dimensions, inducing exactly the same data dependences. Despite this, the message passing Fortran code treats the non-distributed $k$ dimension much differently than the distributed dimensions. Thus, a conceptually small change, such as pipelining in the $k$ and $j$ dimensions (rather than $j$ and $i$) requires a nearly total rewrite of the code. Inflexible codes are less likely to remain useful in the future. The issue of tile size selection is another problem with the message passing code. The task is left entirely to the programmer, for the compiler has no concept of the communication/computation tradeoff that guides tile size selection in pipelined codes. The programmer can either experimentally find the optimal tile size, but this limits the future value of the code, for changes to the code or the hardware on which it runs requires that new tile sizes be calculated. Alternatively, the programmer can implement an automatic tile size selection algorithm [LJ99], but these codes must be integrated into the core computation, further obscuring its logic.

### *6.3.3 Comparison*

The next section will answer the question of how well the A–ZPL compiler addresses the performance concerns cited above, so here we consider the issue of usability. The A–ZPL implementation scores well in every regard that the message passing Fortran code falls short: portability, complexity, asymmetry, and tile size selection.

While the message passing code is bound to a single, fixed communication paradigm (in this case, message passing), the A–ZPL code is not. A–ZPL provides high-level language abstractions that represent the common forms of abstract data motion, shielding the compiler from machine-specific implementation details. The A–ZPL compiler and run-time system are free to exploit what-

ever paradigm best utilizes each machine[CCS98]. As a result, A–ZPL programs are more portable than fixed paradigm programs such as the message passing SWEEP3D.

A–ZPL abstractions not only enhance portability, they reduce complexity and programmer effort. The code fragments in Figures 6.3 and 6.4 clearly illustrate this point. A programmer familiar with both languages will certainly find the A–ZPL easier to write and understand than the message passing. Line counts emphasize this point: 292 for the message passing versus 115 for the A–ZPL. Programmers need not be concerned with the details of implementing mundane and well understood mechanisms (in this case, pipelined parallelism), for the compiler is well equipped to do it for them. This reduction in complexity enhances maintainability and, with portability, prolongs the life of software, drastically reducing its cost.

Asymmetries arise in message passing codes because the programmer customizes a program to a particular goal on a particular machine. For example, the message passing SWEEP3D code assumes that only the *i* and *j* dimensions of the problem are distributed. Perhaps some machines would benefit from a different distribution, but it is impractical to even experiment with alternatives because this would require a complete rewrite. This is trivial when using A–ZPL, because the compiler is responsible for the tedious details of implementing the language's abstractions. Programmers may attempt to avoid these asymmetries by writing their codes in as general a way as possible, but this kind of generality usually comes at the expense of added complexity.

We have already enumerated the problems concerning tile size selection for a message passing implementation of SWEEP3D. A–ZPL does not suffer from these problems because the compiler has a complete view of the pipelining process; thus, it is well equipped to automatically select an appropriate tile size using one of the existing techniques [LJ99].

### *6.4 Performance*

We evaluate the performance of the message passing Fortran and A–ZPL implementations of SWEEP3D on the Cray T3E and the IBM SP. Before we examine final performance, we first consider the issue of tile size selection.

### *6.4.1 Tile Size*

Tile size balances the tradeoff between parallelism and communication, thus tile size selection is an important factor in determining performance of pipelined parallel wavefronts. We have argued that the high-level representation of A–ZPL enables the compiler to automatically select the tile size, but we have not yet implemented this. For this reason, we will experimentally find the optimal tile size for both the Fortran and A–ZPL codes.

Adjustment to the tile size (*i.e.*, variation of *MK* and *MMI*) balance the tradeoff between communication and computation in the pipeline. This raises the question of how sensitive performance is to tile size. For both the message passing Fortran code and the A–ZPL, we found the optimal tile size for each problem size and machine by exhaustively searching the space of all possible tile sizes. Figure 6.5 contains a representative of the data used to determine optimal tiles. This data is for a $50 \times 50 \times 50 \times 6$ problem size and running on four Cray T3E processors.

Both the Fortran and A–ZPL figures clearly demonstrate the communication/computation tradeoff. First, consider the Fortran case in Figure 6.5(a). It is clear that the smallest execution time for a particular value of *MMI* is when $MMI \times MK = 6$. An increase or decrease in *MK* from this point leads to a corresponding increase in execution time from lost parallelism or increased communication overhead, respectively. The A–ZPL data in Figure 6.5(b) is analogous, though somewhat more subtle.

Furthermore, the Fortran times exhibit a general trend of decreasing as *MMI* increases. This behavior arises because an increase of *MMI* implies a reduction of the tile size along the angular dimension, resulting in the outer loop iterating fewer times. When $MMI = 6$, the outer loop iterates over only a single tile, thus minimizing the overhead associated with this loop. The A–ZPL code suffers less from this problem because it has lower overhead associated with this loop as a result of the fact that the initialization of `phikb` appears outside the main loop. Although this requires that `phikb` be larger than it would otherwise be, it reduces sensitivity to *MMI*. In addition, the A–ZPL code optimizes array references across all of the code (tiling and inner loops), whereas it is likely the Fortran is only able to optimize across the inner loops.

In summary, the high-level representation of A–ZPL enables more efficient loop generation, thus reducing the sensitivity of performance to tile size.

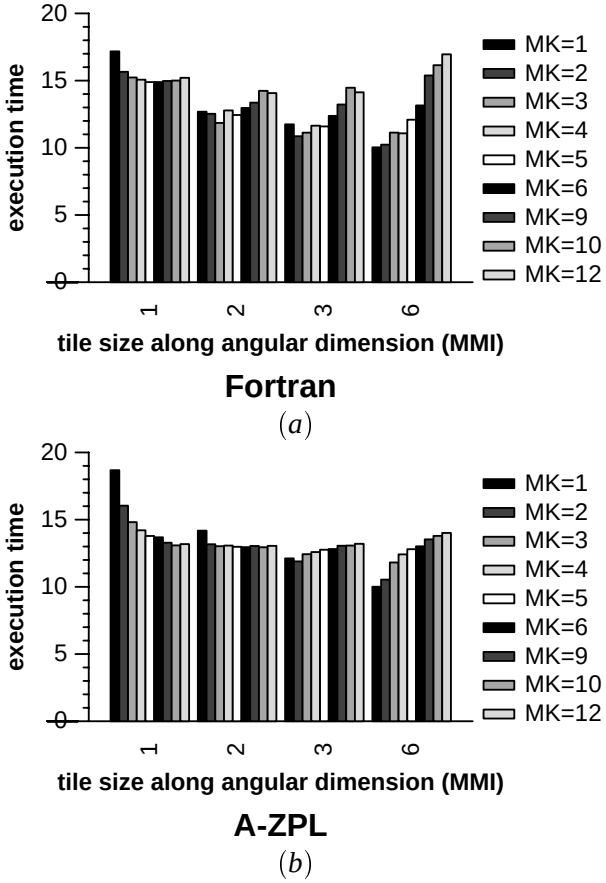



Figure 6.5: Impact of tile size on performance for SWEEP3D implemented in (a) message passing Fortran and (b) A–ZPL. The x-axis represents variations in the tile size along the angular dimension (*MMI*) and each bar represents a variation in the tile size along the *k* spatial dimension (*MK*). This data is for a $50 \times 50 \times 50 \times 6$ problem size and running on four Cray T3E processors.

### *6.4.2 Running Time*

In this section we examine the ultimate performance of Fortran and A–ZPL implementations of SWEEP3D. Each reported execution time is the smallest of all tile size configurations, and for each tile configuration, each time is the smallest of at least three identical trials. Variations were minimal, so we do not report them.

Figure 6.6 presents execution-time data on the Cray T3E and IBM SP for three problem sizes. The white and black bars represent Fortran and A–ZPL performance, respectively. Execution time

is scaled to that of Fortran. Thus a bar less than 1.0 is faster than Fortran. Because the Cray Fortran compiler includes a library substitution optimization, we include a third gray bar, representing Fortran performance when library substitution is disabled. Library substitution performs pattern matching to replace particular code idioms with function calls to highly optimized libraries (in this case, the level 2 BLAS `SGEMV` and `SGER`). Thus the white bar represents Fortran performance with library substitution and the gray bar represents performance without.

First, we examine the two Cray T3E Fortran bars. This data clearly shows that the benefit of library substitution is a function of problem size. For small problems, the overhead of calling into a library swamps the benefit of the library. This fact manifests itself it two ways: *(i)* the Fortran code without library substitution becomes relatively worse as the problem size grows, and *(ii)* the Fortran without library substitution becomes relatively better as the number of processors increases (*i.e.*, the amount of data per processor decreases). Furthermore, neither bar is consistently better than the other. Part of the reason for this is that the libraries solve a two-dimensional problem that represents a portion of a four-dimensional problem. As a result, the two-dimensional problems are unusually small. In fact, the largest problem size in Figure 6.6 is a very large problem, yet library substitution shows minimal benefit. For this reason, this code will have better general behavior without library substitution.

Next, we consider the A–ZPL data. Because tiling is not necessary for a single processor, all the codes make identical use of memory in this case, and they have comparable cache behavior. For a single processor, A–ZPL demonstrates superior scalar performance. Again, this arises from optimizing entire loop nests rather than just the inner loops. Now we examine variations in the number of processors. As the number of processors increases, the amount of data per processor decreases, resulting in two opposing effects. On one hand, less data results in greater loop overhead, and the A–ZPL has relatively greater loop overhead than the Fortran because entire loops are optimized. This behavior is most clear in the smallest problem size. On the other hand, less data means that the memory (and cache) deficits of the A–ZPL decrease. Thus, as the number of processors increases for the largest problem size, the relative performance of the A–ZPL improves. The medium-sized problem shows these two factors competing. The relative A–ZPL performance decreases from four to 16 processors and increases from 16 to 64. Another general trend is that the relative A–ZPL performance decreases as the problem size grows. Again, this comes from its poorer use of memory.

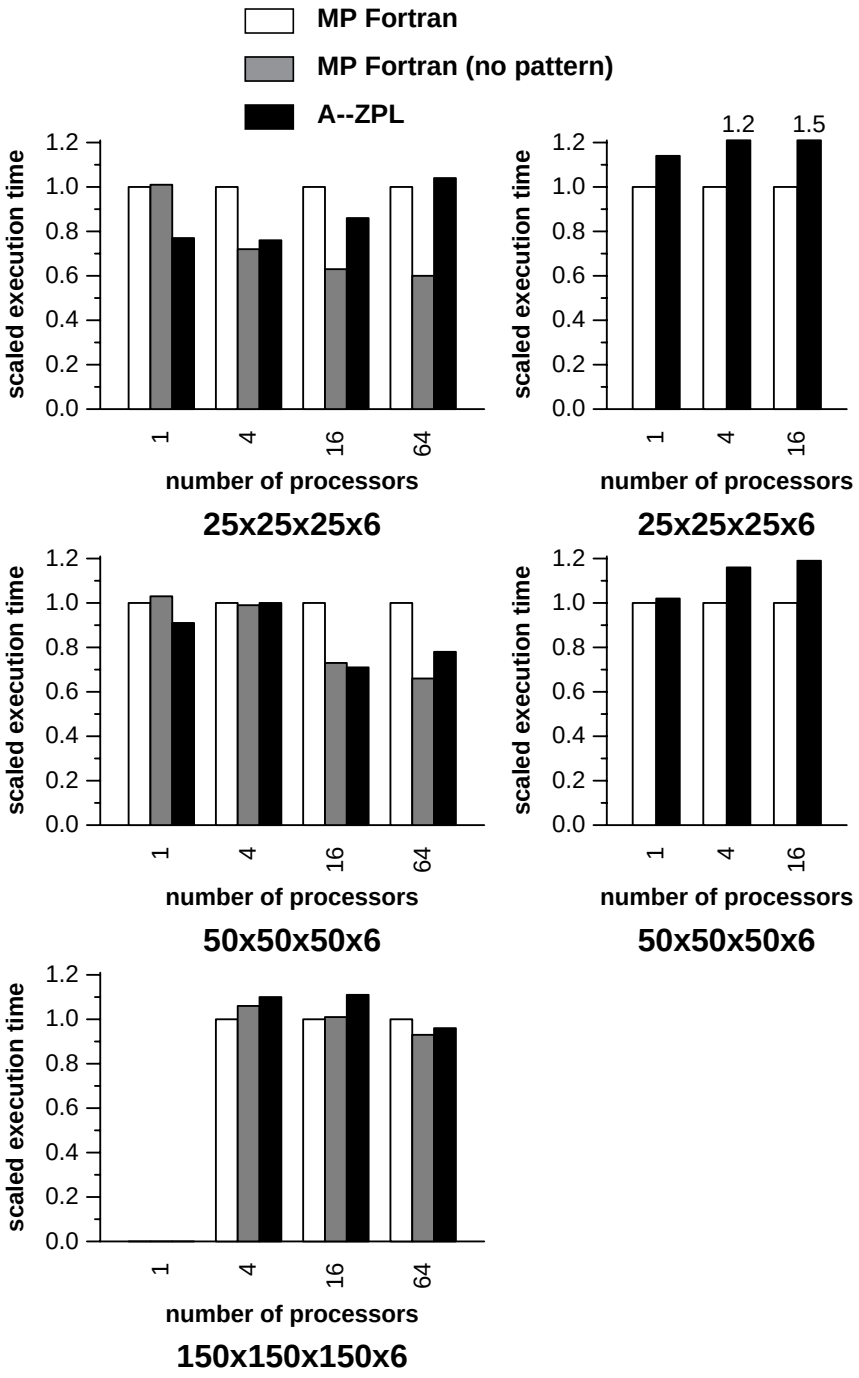



Figure 6.6: Scaled execution time of A–ZPL versus message passing Fortran on the (a) Cray T3E and (b) IBM SP. The label under each graph indicates the problem size. Note that $150 \times 150 \times 150 \times 5$ problems are too large to run on a single processor of the T3E, and it can not be run at all on our SP configuration.

In summary, it is clear that despite subtle differences, the performance of the A–ZPL codes competes with—and sometimes outperforms—the highly optimized Fortran codes. The goal was to compete, but it turns out that the high-level representation can actually result in superior code.

### *6.5 Summary*

We have evaluated several representations of the SWEEP3D benchmark. We find that an automatic parallelization implementation does not result in consistently efficient code, for different compilers use different optimization strategies and the optimization is brittle.

The message passing and A–ZPL implementation both perform well. The message passing code requires that the programmer manage every detail of the implementation, resulting in a large code that obscures the fundamental computation that it performs. Assembly language programs are the serial analog of this situation, *i.e.*, the message passing programmer must manage issues analogous to register allocation and instruction scheduling, tasks best left to a compiler.

The A–ZPL code, like all A–ZPL programs, does not require the programmer to manage implementation details. Instead the language defines abstractions that permit the programmer to reason about the primary performance implications (*e.g.*, parallelism) of a code, and the compiler manages the details of mapping these abstractions to a particular machine. The programmer is well aware of the parallelism implicit in array operations and the pipelined parallelism arising from the primed at references, but it is the compiler's responsibility to manage the interprocessor communication, the tiling to set up the pipelining, and the array contraction necessary for efficient use of memory and the data cache. The result is a program that has portable performance and is easy to develop and maintain.

# Chapter 7

# CONCLUSIONS

## *7.1 Summary and Contributions*

Despite the existence of powerful parallel machines, the promise of their peak performance and enthusiasm for their use, exploiting parallelism is still impractical for most users. This is because performance typically comes at the expense of portability (*i.e.*, performance portability), usability, or both; and most potential users are unwilling or unable to accept this trade-off. Until parallel programming systems perform well, are portable, and usable, parallelism will be of little value to most users.

We address this problem with the integrative design of language-level abstractions, compiler technology, and predictive performance models. Roughly speaking, language-level abstractions address the usability goal of simplification. With well designed or chosen abstractions, the programmer is presented with a computational view that hides complex details. It is then the role of the compiler to map these abstractions to a machine, specializing them to a particular architecture. A tension between performance and usability persists in the development of abstractions and compiler technology; thus, their design must be guided by a portable predictive performance model.

Just as fundamental parallel properties are made apparent to programmers, secondary details must be abstracted in support of usability. Tedious, yet tractable, details of parallel programming are relegated to the compiler. Thus, a programmer and compiler have specific and distinct roles. We call the division of labor between programmer and compiler *programmer-compiler separation*, for it determines the responsibilities of each entity.

- The *programmer* is responsible for abstract parallel and sequential algorithmic issues.
- The *compiler* manages the tractable elements of mapping abstract representations to a particular machine.

The design of the Advanced–ZPL (A–ZPL) parallel programming language was guided by this

programmer-compiler separation. The A–ZPL language abstractions permit programmers to reason about the parallel and locality implications of their code, for the abstractions are subject to a portable predictive performance model. At the same time they are sufficiently abstract and intuitive that the language is easy to use as compared to the alternatives.

We have examined portions of the A–ZPL language and its compiler and evaluate them in the light of the above programmer-compiler separation. We have considered elements of the language that demonstrate the role of programmer-compiler separation. On one hand, we have examined a complex task that is entirely managed by the compiler: the generation of loop nest from array statements. On the other hand, we have considered a circumstance where new language abstractions dramatically improve the efficacy of the language: support for pipelining wavefront computations.

Loop generation is a performance-critical aspect of array language compilation. Effective loop generation in A–ZPL consists of two components, statement fusion and array contraction. *Statement fusion* is analogous to loop fusion except that it is performed by the compiler on array statements before they have been converted to scalar loop nests. *Array contraction* is a program transformation enabled by statement fusion that permits a single scalar value to be used in place of an array. Together, these optimizations dramatically improve data cache behavior and performance (frequently by 25% and in some cases by a factor of 4) in most programs. In addition, array contraction reduces memory consumption, permitting larger problems to be solved in a fixed-sized memory. We argue that the task of loop generation is eminently tractable for the compiler, and our experiments support this claim.

Wavefront computations occur frequently, for example in solvers and dynamic programming codes. We show fully automatic approaches to parallelizing such code are impractical. We describe the design and implementation of a general language abstraction that consistently admits efficient pipelined parallel implementations of wavefront computations. We show that this approach consistently performs better than the alternatives.

This dissertation is an in-depth study of the development of language-level abstractions and compiler technology guided by portable predictive performance models. All models are well defined, and all abstractions and compiler techniques are implemented and evaluated in the context of a complete practical programming language.

The work in this dissertation makes the following contributions.

- It illustrates programmer-compiler separation via the A–ZPL parallel programming language.
- It motivates and justifies particular A–ZPL language design decisions in support of programmer-compiler separation considerations.
- It presents and experimentally evaluates unique techniques for array contraction.
- It introduces a novel language abstraction for representing wavefront computations for pipelined parallel execution.
- It quantitatively and qualitatively evaluates this new abstraction via kernels computations and large applications.
- And it describes core elements of the A–ZPL compilation process, including efficient loop generation for array statements.

### *7.2 Future Work*

The work described in this dissertation is mature and fully implemented. Nevertheless, there are many avenues for future research in the short-, medium-, and long-term.

In the short-term, details of pipelining warrant additional study. For example, we must implement a scheme for dynamic tile size selection. Will existing approaches suffice, or can we exploit the static character of A–ZPL programs? Furthermore, tile size is not the only parameter that may be customized for particular machines. Specialization of other parameters, such as communication mechanism, deserves further study. In addition, we will study the implications of skewing the iteration space to improve parallelism. Finally, we have assumed that all array dimensions are potentially distributed, limiting the variety of wavefronts that may be practically implemented. For this reason, we will consider mechanisms by which some dimensions are statically identified as non-distributed.

In the medium-term, the A–ZPL language must be completed, resulting in a general purpose parallel programming language. We seek a unified representation that brings together task/data parallelism, control/data irregularity, computation/data distribution, and load balancing. This is a tall order, but we believe our success in the data parallel domain will serve us well. We may begin with the study of the generalization of pipelining moving us closer to task parallelism. We believe the A–ZPL region, or an analogous structure, will serve as the tool for both manipulating parallelism and reasoning about parallel performance, as in A–ZPL. If the completed version of A–ZPL does, in fact, provide performance, usability, and portability, we will explore its use as a portable parallel

intermediate language. Such a system could encourage the development of even higher level parallel programming systems.

Another medium-term project entails using the A–ZPL representation in other parallel domains. For example, most modern processors provide fine-grain parallel operations for multi-media applications. Currently, it is not clear how programs should be written to exploit these hardware features. We believe that a less restricted form of A–ZPL would address this problem. Perhaps we will investigate the augmentation of a popular language like C++ or Java with A–ZPL abstractions, such as regions and array operators.

In the long-term, we would like to do language design with programmer-compiler separation in other domains. For example, consider low-power devices. Naturally, software will implement the bulk of the functionality of such devices, but the power consumption characteristics of codes are not manifest in the source program. Programmer-compiler separation guided by a power performance model would be a great asset in creating development tools.

## BIBLIOGRAPHY

[ABLZ99] W. Amme, P. Braun, W. Löwe, and E. Zehendner. LogP modelling of list algorithms. In *Proceedings 11th Symposium on Computer Architecture and High Performance Computing*, pages 157–63, 1999.

[ABM+92] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*. McGraw-Hill, New York, NY, 1992.

[ABR96] Rumen Andonov, Hafid Bourzoufi, and Sanjay Rajopadhye. Two-dimensional orthogonal tiling: from theory to practice. In *IEEE International Conference on High Performance Computing*, pages 225–231, December 1996.

[Acca] Accelerated Strategic Computing Initiative. ASCI homepage. *http://www.lanl.gov/projects/asci/*.

[Accb] Accelerated Strategic Computing Initiative. ASCI SWEEP3D homepage. *http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/sweep3d_readme.html*.

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[AGNS90] Gail A. Alverson, William G. Griswold, David Notkin, and Lawrence Snyder. A flexible communication abstraction for nonshared memory parallel computing. In *Proceedings of Supercomputing '90*, pages 584–93, 1990.

[Akl89] Selim G. Akl. *The design and analysis of parallel algorithms*. Prentice Hall, Englewood Cliffs, N.J., 1989.

[AR97] Rumen Andonov and Sanjay Rajopadhye. Optimal orthogonal tiling of 2-d iterations. *Journal of Parallel and Distributed Computing*, 45(20):159–165, September 1997.

[ARY98] Rumen Andonov, Sanjay Rajopadhye, and Nicola Yanev. Optimal orthogonal tiling. In *Proceedings of Euro-Par'98*, pages 480–490, December 1998.

[AS91] Richard J. Anderson and Lawrence Snyder. A comparison of shared and nonshared memory models of parallel computation. *Proceedings of the IEEE*, 79(4):480–7, April 1991.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Priciples, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1988.

[AWMC+95] Vikram S. Adve, Jhy-Chun Wang, John Mellor-Crummey, Daniel A. Reed, Mark Anderson, and Ken Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Supercomputing '95*, December 1995.

[BB82] D.H. Ballard and C.M. Brown. *Computer Vision*. Prentice-Hall, 1982.

[BBB+94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks (94). Technical report, RNR Technical Report RNR-94-007, March 1994.

[BD94] Adam Beguelin and Jack Dongarra. *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1994.

[BHS+95] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical report, NAS Report NAS-95-020, December 1995.

[BK94] Ray Barriuso and Allen Knies. SHMEM user's guide for C. Technical report, Cray Research, Inc., 1994.

[BL99] Karthik Balasubramanian and David K. Lowenthal. Efficient support for pipelining in distributed shared memory systems. Submitted for publication., 1999.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA, 1990.

[Ble92] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, January 1992.

[Ble96] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[CCL+96] B. Chamberlain, S. Choi, E Lewis, C. Lin, L. Snyder, and W. D. Weathersby. Factor-join: A unique approach to compiling array languages for parallel machines. In David Sehr, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Proceedings of the Ninth International Workshop on Languages and Compilers for Parallel Computing*, pages 481–500. Springer-Verlag, August 1996.

[CCL+98a] Bradford Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. The case for high level parallel programming in zpl. *IEEE Computational Science and Engineering*, 5(3):76–86, July–September 1998.

[CCL+98b] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL's WYSIWYG performance model. In *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 50–61. IEEE Computer Society Press, March 1998.

[CCL+00] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering*, 26(3):197–211, March 2000.

[CCS98] Bradford Chamberlain, Sung-Eun Choi, and Lawrence Snyder. A compiler abstraction for machine independent parallel communication generation. In *Tenth International Workshop on Languages and Compilers for Parallel Computing*, pages 261–76, Berlin, Germany, 1998. Springer–Verlag.

[CDS00] Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *SC200*, 2000.

[Cha00] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Programming Language*. PhD thesis, University of Washington, 2000.

[CHL78] W. Crowley, C. P. Hendrickson, and T. I. Luby. The SIMPLE code. Technical Report UCID-17715, Lawrence Livermore Laboratory, 1978.

[Cho99] Sung-Eun Choi. *Machine independent communication optimization*. PhD thesis, University of Washington, 1999.

[CK94] Steve Carr and Ken Kennedy. Scalar replacement in the presence of conditional control flow. *Software – Practice and Experience*, 24(1):51–77, January 1994.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eiken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth Symposium on Principle and Practice of Parallel Programming*, pages 1–12, May 1993.

[CKPN00] Junwei Cao, Darren J. Kerbyson, Efstathios Papaefstathiou, and Graham R. Nudd. Performance modeling of parallel and distributed computing using PACE. In *Proceedings of the 2000 IEEE International Performance, Computing, and Communications Conference*, pages 485–92, 2000.

[CLLS99] Bradford L. Chamberlain, E Christopher Lewis, Calvin Lin, and Lawrence Snyder. Regions: An abstraction for expressing array computation. In *ACM SIGAPL/SIGPLAN International Conference on Array Programming Languages*, pages 41–9, August 1999.

[CLS98] Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. A region-based approach for sparse parallel computing. Technical Report UW-CSE-98-11-01, University of Washington, November 1998.

[CMT94] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improved data locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994. San Jose, CA.

[CS97] Sung-Eun Choi and Lawrenc Snyder. Quantifying the effects of communication optimizations. In *Proceedings of the International Conference on Parallel Processing*, pages 218–222, August 1997.

[CTY94] Ding-Kai Chen, Josep Torrellas, and Pen-Chung Yew. An efficient algorithm for the run-time parallelization of doacross loops. In *Proceedings of the conference on Supercomputing '94*, pages 518–527, November 1994.

[Cyt86] Ron Cytron. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing*, pages 836–44, 1986.

[DCS00] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. A unified framework for eliminating redundant array subexpressions. Submitted for publication., 2000.

[DCSM96] Andrea C. Dusseau, David E. Culler, Klaus Erik Schauser, and Richard P. Martin. Fast parallel sorting under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7(8):791–805, August 1996.

[DLMW95] Marios D. Dikaiakos, Calvin Lin, Daphne Manoussaki, and Diana E. Woodward. The portable parallel implementation of two novel mathematical biology algorithms in ZPL. In *Ninth International Conference on Supercomputing*, 1995.

[DRR96] Frederic Desprez, Pierre Ramet, and Jean Roman. Optimal grain size computation for pipelined algorithms. In *Proceedings of Euro-Par'96*, pages 165–172, 1996.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–8, 1978.

[GMS+95] Manish Gupta, Sam Midkiff, Edith Schonberg, Ven Seshadri, David Shields, Ko-Yang Wang, Wai-Mee Ching, and Ton Ngo. An HPF compiler for the IBM SP2. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference (CD-ROM)*, 1995.

[GOST92] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Proceedings of the Fifth International Workshop on Languages and Compilers for Parallel Computing*, pages 281–295. Springer-Verlag, 1992.

[GS90] Raymond Greenlaw and Lawrence Snyder. Achieving speedups for APL on an SIMD distributed memory machine. *International Journal of Parallel Programming*, 19(2):111–27, April 1990.

[Hig97] High Performance Fortran Forum. *High Performance Fortran Langauge Specification, Version 2.0*. January 1997.

[Hig00] High Performance Technical Computing Group. Alphaserver SC: Scalable supercomputing. Technical Report 1234-0400A-WWEN, Compaq Computer Corporation, July 2000.

[HKT91] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Supercomputing '91*, pages 96–100, Albuquerque, NM, November 1991.

[HKT92] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *International Conference on Supercomputing*, pages 1–14, Washington, DC, July 1992.

[HLJ95] Gwan Hwan Hwang, Jenq Keun Lee, and Dz Ching Ju. An array operation synthesis scheme to optimize Fortran 90 programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[JGB97] E. Johnson, D. Gannon, and P. Beckman. HPC++: Experiments with the parallel standard template library. In *Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, pages 124–131, July 1997.

[Ju92] Dz-Ching Ju. *The Optimization and Parallelization of Array Language Programs*. PhD thesis, University of Texas–Austin, August 1992.

[KB94] Scott R. Kohn and Scott B. Baden. A robust parallel programming model for dynamic non-uniform scientific computations. Technical Report CS94-354, University of California, San Diego, Dept. of Computer Science and Engineering, March 1994.

[KBA92] K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the first-order form of three-dimensional discrete ordinates equations on a massively parallel machine. *Transactions of the American Nuclear Society*, 65:198–9, 1992.

[KM92] Ken Kennedy and Kathryn S. McKinley. Optimizing for parallelism and data locality. In *International Conference on Supercomputing*, pages 323–334, July 1992.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.

[LHZ98] Honghui Lu, Y. Charlie Hu, and Willy Zwaenepoel. OpenMP on networks of workstations. In *Proceedings of ACM/IEEE SC98*, 1998.

[Lin92] Calvin Lin. *The portability of parallel programs across MIMD computers*. PhD thesis, University of Washington, 1992.

[LJ99] David K. Lowenthal and Michael James. Run-time selection of block size in pipelined parallel programs. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 82–7, 1999.

[LLS92] J. Lee, Calvin Lin, and Lawrence Snyder. Programming SIMPLE for parallel portability. In *Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 84–98, Berlin, Germany, 1992. Springer–Verlag.

[LLS98] E Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–59, June 1998.

[LLST95] E Christopher Lewis, Calvin Lin, Lawrence Snyder, and George Turkiyyah. A portable parallel n-body solver. In D. Bailey, P. Bjorstad, J. Gilbert, M. Mascagni, R. Schreiber, H. Simon, V. Torczon, and L. Watson, editors, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 331–336. SIAM, 1995.

[Low00] Douglas Low. Storage conflict elimination in a parallel array language. In preparation., 2000.

[LS91] Calvin Lin and Lawrence Snyder. A portable implementation of SIMPLE. *International Journal of Parallel Programming*, 20(5):363–401, October 1991.

[LS92] Calvin Lin and Lawrence Snyder. Portable parallel programming: cross machine comparisons for SIMPLE. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 564–9, 1992.

[LS93] Calvin Lin and Lawrence Snyder. Data ensembles in Orca C. In *Fifth International Workshop on Languages and Compilers for Parallel Computing*, pages 112–23, Berlin, Germany, 1993. Springer–Verlag.

[LS94a] Calvin Lin and Lawrence Snyder. SIMPLE performance results in ZPL. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Workshop on Languages and Compilers for Parallel Computing*, pages 361–375. Springer-Verlag, 1994.

[LS94b] Calvin Lin and Lawrence Snyder. ZPL: an array sublanguage. In *Sixth International Workshop on Languages and Compilers for Parallel Computing*, pages 96–114, Berlin, Germany, 1994. Springer–Verlag.

[LS95] Calvin Lin and Lawrence Snyder. SIMPLE performance results in ZPL. In *Seventh International Workshop on Languages and Compilers for Parallel Computing*, pages 361–75, Berlin, Germany, 1995. Springer–Verlag.

[LSA+94] C. Lin, L. Snyder, R. Anderson, B. Chamberlain, S. Choi, G. Forman, E. Lewis, and W. D. Weathersby. ZPL vs. HPF: A comparison of performance and programming style. Technical Report 95–11–05, Department of Computer Science and Engineering, University of Washington, 1994.

[MA97] Naraig Manjikian and Tarek S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE transactions on parallel and distributed systems*, 8(2):193–209, February 1997.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, California, 1997.

[Ngo97] Ton Anh Ngo. *The role of performance models in parallel programming and languages*. PhD thesis, University of Washington, 1997.

[NS92] Ton A Ngo and Lawrence Snyder. On the influence of programming models on shared memory computer performance. In *Proceedings of Scalable High Performance Computing Conference SHPCC-92*, pages 284–91, 1992.

[NSC97] Ton A. Ngo, Lawrence Snyder, and Bradford L. Chamberlain. Portable performance of data parallel languages. In *SC97: High Performance Networking and Computing*, November 1997.

[NSS+88] David Notkin, Lawrence Snyder, David Socha, Mary L. Bailey, Bruce Forstall, Kevin Gates, Raymond Greenlaw, William G. Griswold, Thomas J. Holman, Richard Korry, Gemini Lasswell, , Robert Mitchell, and Philip A. Nelson. Experiences with poker. In *Proceedings of the ACM/SIGPLAN PPEALS, Parallel Programming: Experience with Applications, Languages and Systems*, pages 10–20, 1988.

[OSKO95] Hiroshi Ohta, Tasuhiko Saito, Masahiro Kainaga, and Hiroyuki Ono. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *International Conference on Supercomputing*, pages 270–9, Barcelona, Spain, 1995.

[PZ96] Terrence Pratt and Marvin Zelkowitz. *Programming Languages: Design and Implementation, 3rd Edition*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[Ran82] Brian Randell. *Origins of Digital Computers: Selected Papers*. Springer-Verlag, Berlin, 1982.

[RBS96] Wilkey Richardson, Mary Bailey, and William H. Sanders. Using ZPL to develop a parallel Chaos router simulator. In *1996 Winter Simulation Conference*, pages 806–16, December 1996.

[RK96] Gerald Roth and Ken Kennedy. Dependence analysis of Fortran90 array syntax. In *Proceedings of the Int'l Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, August 1996.

[RP89] Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 69–80, June 1989.

[SG91] Vivek Sarkar and Guang R. Gao. Optimization of array accesses by collective loop transformations. In *International Conference on Supercomputing*, pages 194–205, June 1991.

[SLY90] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[Sny81] Lawrence Snyder. Overview of the CHiP computer. In *Proceedings of the First International Conference on Very Large Scale Integration*, pages 237–46, 1981.

[Sny83] Lawrence Snyder. Introduction to the poker parallel programming environment. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 289–92, 1983.

[Sny84] Lawrence Snyder. Parallel programming and the poker programming environment. *Computer*, 17(7):27–36, July 1984.

[Sny86] Lawrence Snyder. Type architectures, shared memory, and the corollary of modest potential. *Annual review of computer science*, 1:289–317, 1986.

[Sny90] Lawrence Snyder. The XYZ abstraction levels of Poker-like languages. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Second Workshop on Parallel Compilers and Algorithms*, pages 470–89. MIT Press, 1990.

[Sny92] Lawrence Snyder. Critique of the Poker parallel programming environment. In *Proceedings of the International Conference on Parallel Computing*, pages 419–26, 1992.

[Sny93] Lawrence Snyder. Foundations of practical parallel programming languages. In *Proceedings of the Second International Conference of the Austrian Center for Parallel Computation*, pages 115–34. Springer-Verlag, 1993.

[Sny99] Lawrence Snyder. *The ZPL Programmer's Guide*. The MIT Press, 1999.

[Soc91] David G. Socha. *Supporting fine-grain computation on distributed memory parallel computers*. PhD thesis, University of Washington, 1991.

[SOHL+98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI—The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 2nd edition, 1998.

[SS86] Lawrence Snyder and David Socha. Poker on the cosmic cube: the first retargetable parallel programming language and environment. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 628–35, 1986.

[SSV99] David Sundaram-Stukel and Mark K. Vernon. Predictive analysis of a wavefront application using LogGP. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1999.

[Sta] Standard Performance Evaluation Corporation. SPEC homepage. *http://www.spec.org*.

[SY91] Hong-Men Su and Pen-Chung Yew. Efficient doacross execution on distributed shared-memory multiprocessors. In *Proceedings of the 1991 conference on Supercomputing*, pages 842–853, November 1991.

[Thi91] *C* Programming Guide, Version 6.0.2*. Thinking Machines Corporation, Cambridge, Massachusetts, June 1991.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–11, August 1990.

[vN45] John von Neumann. First draft of a report on the EDVAC. Technical Report W-670-ORD-492, Moore School of Electrical Engineering, University of Pennsylvania, June 1945. Reprinted (in part) in Randell [Ran82, pp. 382–92].

[WGS00] A. J. Wagner, L. Giraud, and C. E. Scott. Simulation of a cusped bubble rising in a viscoelastic fluid with a new numerical method. *Computer Physics Communications*, 129(1–3):227–32, July 2000.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN'91 Conference on Program Language Design and Implementation*, June 1991.

[Wol96] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[ZPL] ZPL Project. ZPL project homepage. *http:/www.cs.washington.edu/research/zpl*.

# Vitae

E Christopher Lewis was born in upstate New York (May 1970) and has since lived in Massachusetts, Pennsylvania, Vermont, North Carolina, New York (again), Washington, and Pennsylvania (again). He has attended Cornell University (B.S. 1993) and the University of Washington (M.S. 1996).